Puma

User's Manual

Matthias Urban

March 11, 2009

Contents

1 About

This is the user documentation of the Puma library. Puma is an extensible C/C++ parser and code transformation library written in C++. It provides the following key features:

- Built-in C preprocessor with separate preprocessor syntax tree
- Lexical analysis of C and C++ source code providing separate token chains
- Syntactic analysis of token chains providing separate syntax trees
- Semantic analysis of syntax trees providing separate semantic information databases
- Source code transformation on token and syntax tree level
- ISO/IEC 9899-1999(E) - C conformance
- ISO/IEC 14882:1998(E) - C++ conformance

Puma is based on a top-down parser implementation that makes it easy to add own extensions to the parser. There are already some non-standard extensions implemented, i.e. GNU C/C++ and VisualC++ extensions. These extensions are introduced into the parser by aspects using AspectC++. This is optional and requires an installed AspectC++ compiler.

1.1 License

Puma is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

2 Installation

2.1 Getting Puma

The source code of the Puma library is available together with the AspectC++ source package from the *AspectC++ Homepage*, or via SVN with the following command:

```
svn checkout https://svn.aspectc.org/repos/Puma/trunk Puma
```

2.2 Building the Library

Building Puma is based on GNU make. To build and install the library on Linux without debugging information follow these steps:

1. `cd <PUMA_ROOT_DIRECTORY>`
2. `make TARGET=linux-release`
3. `make TARGET=linux-release install`

The variable `TARGET` specifies the target platform and whether debugging mode is enabled or not. Currently the following values are supported:

`linux`	- Linux debug build
`linux-release`	- Linux release build (Default)
`macosx`	- MacOSX debug build
`macosx-release`	- MacOSX release build
`win32`	- Win32 debug build using mingw32
`win32-release`	- Win32 release build using mingw32

Building Puma for other target platforms may require changes on the file `vars.mk` in the root directory of Puma.

Additional build and compilation flags can be specified using the following variables.

`CPP_OPTFLAGS`	- Target compiler flags
`AC_OPTFLAGS`	- AspectC++ compiler flags

The following make targets are available.

`all`	Default target. Build the library. Does not build the examples and the doxygen documentation.
`weave`	Generate the woven library sources. Applies all active extensions. Needs an installed AspectC++ compiler.
`compile`	Compile the woven library sources.
`install`	Install the built library to `/usr/local`. The install location can be changed by setting the variable `PREFIX` (e.g. `make PREFIX=$HOME/usr install`).
`uninstall`	Uninstall the library.
`examples`	Build the example Puma applications in the `examples` directory.
`examples-clean`	Remove temporary build files from the example Puma applications.
`doxygen`	Generate a doxygen documentation for the Puma library. Requires that doxygen is installed.
`showinfo`	Show the compiler and linker options used to build the library.
`clean`	Remove the generated and temporary build files. Does not clean the tools and examples.
`cleanall`	Same as `clean` but also cleans the tools and examples.
`libclean`	Remove the temporary build files. Does not remove the generated/woven files.
`distclean`	Same as `cleanall`.
`tools`	Build the tools in the `tools` directory.
`tools-clean`	Remove temporary build files from the `tools` directory.
`test`	Run the tests in the `tests` directory.
`test-clean`	Remove temporary test result files from the `tests` directory.

2.2.1 Building Extensions

The Puma library can be build with some extensions. These extensions are defined in the file `extensions.mk` in the Puma root directory.

`gnuext`	- GNU C/C++ language extensions
`winext`	- VisualC++ language extensions
`acppext`	- AspectC++ language extensions
`tracing`	- Syntax rule tracing
`matchexpr`	- AST match expressions

The extensions to be included can be specified by setting the variable `EXTENSIONS`.

Example:

```
make EXTENSIONS="gnuext tracing"
```

This command builds the library including GNU C/C++ language extensions and tracing.

3 Using the Library

3.1 Puma Namespace

The classes in the Puma library are enclosed in the namespace Puma. An application may either add a

```
using namespace Puma;
```

statement before using Puma classes or use full qualified names for referencing Puma classes, e.g.

```
Puma::Token* token = 0;
```

3.2 Compiling and Linking

A Puma application is usually compiled and linked with the following compiler options:

```
-I$PUMA/include -L$PUMA/lib -lpuma
```

3.3 Configuration Options

Several aspects of the functionality of Puma can be configured using command line options or a configuration file. The following configuration options are understood by the library.

Preprocessor Options:

-A <PREDICATE>(<ANSWER>)	Define a preprocessor assertion, e.g. -Asystem(linux)
-D <NAME>[=<BODY>]	Define a preprocessor macro, e.g. -DSYSTEM=linux, -D DEBUG
--lock-macro <NAME>[=<BODY>]	Define a preprocessor macro that cannot be redefined

`-U <NAME>`	Undefine a preprocessor macro, e.g. `-U SYSTEM`
`--inhibit-macro <NAME>`	Prevent a preprocessor macro from being defined
`-I <PATH>`	Add another include path
`--include <FILE>`	Include the given file in every translation unit

Parser Options:

`--skip-bodies-all`	Do not parse function bodies
`--skip-bodies-tpl`	Do not parse function bodies of templates
`--skip-bodies-non-prj`	Do not parse function bodies in non-project files
`--size-type <TYPE>`	Set the internal type for `size_t`
`--ptrdiff-type <TYPE>`	Set the internal type for `ptrdiff_t`
`--real-instances`	Enable real template instantiation
`--match-expr`	Enable match expression language extensions
`--lang-c`	Set the language for input files to C
`--lang-ec++`	Set the language for input files to EC++
`--lang-c++`	Set the language for input files to C++
`--lang-ac++`	Set the language for input files to AC++

VisualC++ Extension Options:

`--vc`	Enable VisualC++ language extensions
`--import-handler <FILE>`	Set a handler for resolving `#import` directives

GNU C/C++ Extension Options:

`--gnu`	Enable all GNU C/C++ language extensions
`--gnu-2.95`	Enable GNU C/C++ 2.95 language extensions
`--gnu-nested-fct`	Enable GNU C/C++ nested functions
`--gnu-condition-scope`	Enable GNU C/C++ condition scope
`--gnu-param-decl`	Enable GNU C/C++ parameter declarator
`--gnu-fct-decl`	Enable GNU C/C++ function declarator

--gnu-param-scope	Enable GNU C/C++ function parameter scope
--gnu-default-args	Enable GNU C/C++ function default arguments
--gnu-extended-asm	Enable GNU C/C++ extended asm syntax
--gnu-extended-expr	Enable GNU C/C++ extended expressions
--gnu-long-long	Enable GNU C/C++ long long type
--gnu-name-scope	Enable GNU C/C++ name scope
--gnu-fct-attribute	Enable GNU C/C++ function attributes
--gnu-if-then-expr	Enable GNU C/C++ if-then expression syntax
--gnu-std-hack	Enable GNU C/C++ implicit namespace std hack

File Handling Options:

--config <FILE>	Load the given configuration file
-p, --path <PATH>	Add given path as source directory
-d, --dest <PATH>	Add given path as destination directory
-w, --write-protected <PATH>	Add given path as write protected directory
-e, --extension <STRING>	Set the extension for input files
-s, --suffix <STRING>	Set the suffix for saving files
--new-suffix	Replace the old suffix when saving files
--save-overwrite	Overwrite input files when saving
--rename-old	Rename input files when saving using a suffix

3.3.1 Configuration File

All command line options can also be specified in a configuration file.

Each option in the configuration file has to start on a new line.

```
-D i386
-D linux
-I /usr/include
```

Lines beginning with '#' are interpreted as comments and will be ignored.

```
### defines
-D i386
```

```
-D linux
### includes
-I /usr/include
-I /usr/local/include
```

Option arguments containing spaces have to be double-quoted. Double-quotes in the argument have to be escaped.

```
-D __PTRDIFF_TYPE__=int
-D "__SIZE_TYPE__=unsigned int"
-D "__VERSION__=\"4.1.0 (Linux)\""
```

All occurrences of `${Name}` in the configuration file are interpreted as environment variables and replaced by their values, or by nothing if a variable is not defined. To avoid variable replacement `$` has to be escaped.

```
-I ${LIBDIR}/include
-D OS_STR=\"${OSTYPE}\"
### same as: #define OS_STR "linux"
-D OS_VAR=\"\${OSTYPE}\"
### same as: #define OS_VAR "${OSTYPE}"
```

4 File Handling

5 Preprocessor

6 Lexical Analysis

7 Syntactic Analysis

8 Semantic Analysis

9 Code Transformation

10 Reference

10.1 C/C++ Syntax Tree Classes

10.1.1 Semantic Attributes

10.1.1.1 CExprValue

```
#include <Puma/CExprValue.h>
```

Base class for syntax tree nodes representing expressions that can be resolved to a constant value (arithmetic constants and string literals).

10.1.1.2 CConstant

```
#include <Puma/CConstant.h>
```

Semantic information object for arithmetic constants. Derived from *CExprValue*.

10.1.1.3 CSemObject

```
#include <Puma/CSemObject.h>
```

Semantic information for syntax tree nodes referencing objects, classes, or any other entity.

10.1.1.4 CSemValue

```
#include <Puma/CSemValue.h>
```

Semantic information object about values in the syntax tree. Provides the value and type of an expression or entity (name).

10.1.1.5 CSemScope

```
#include <Puma/CSemScope.h>
```

Scope information object for syntax tree nodes. Some syntactic constructs open own scopes, e.g. class definitions, function bodies, and compound statements.

10.1.1.6 CStrLiteral

```
#include <Puma/CStrLiteral.h>
```

String literal abstraction. Holds the string value, its length, and the string type. Derived from *CExprValue*.

10.1.1.7 CWStrLiteral

```
#include <Puma/CWStrLiteral.h>
```

Wide string literal abstraction. Holds the wide string value, its length, and the string type. Derived from *CExprValue*.

10.1.2 Basic Tree Classes

10.1.2.1 CTree

```
#include <Puma/CTree.h>
```

Base class for all C/C++ syntax tree classes.

The syntax tree is the result of the syntactic analysis of the input source code representing its syntactic structure according to the accepted grammar (see class Syntax).

Objects of this class and classes derived from this class are created by the tree builder component of Puma during the parse process. A syntax tree shall be destroyed using the tree builder that has created it by calling its *destroy(CTree*)* method with the root node of the syntax tree as its argument.

The navigation in the syntax tree is done using the methods *Parent()*, *Sons()*, and *Son(int)* const. In a syntax tree "sons" are understood as the syntactic child nodes of a syntax tree node, whereas "daughters" are understood are their semantic child nodes.

Another way to traverse a syntax tree is to implement an own tree visitor based on class *CVisitor*. This is recommended especially for larger syntax trees.

A syntax tree node can be identified by comparing its node name with the node identifier of the expected syntax tree node:

```
if (node->NodeName() == Puma::CT_BinaryExpr::NodeId()) ...
```

Based on the syntax tree further semantic analyses can be performed. Semantic information, like scope, value, type, and object information, is linked into the syntax tree. It can be accessed using the methods *SemScope()*, *SemValue()*, and *SemObject()*. Some nodes provide short-cuts to the semantic type and value information by implementing the methods *Type()* and *Value()*.

The information of the syntax tree can be used to perform high-level transformations of the source code (see class *ManipCommander*).

10.1.2.2 CT_Token

```
#include <Puma/CTree.h>
```

Tree node representing a single token in the source code. Derived from *CTree*.

10.1.2.3 CT_List

```
#include <Puma/CTree.h>
```

Base class for tree nodes representing lists. Derived from *CTree*.

10.1.2.4 CT_Error

```
#include <Puma/CTree.h>
```

Error tree node that is inserted into the tree for syntactic constructs that could not be parsed. Derived from *CTree*.

10.1.2.5 CT_Program

```
#include <Puma/CTree.h>
```

Root node of C/C++ syntax trees. Derived from *CT_DeclList* and *CSemScope*.

10.1.3 Statements

10.1.3.1 CT_Statement

```
#include <Puma/CTree.h>
```

Base class for all tree nodes representing statements. Derived from *CTree*.

10.1.3.2 CT_CmpdStmt

```
#include <Puma/CTree.h>
```

Tree node representing a compound statement. Derived from *CT_List* and *CSemScope*.

10.1.3.3 CT_LabelStmt

```
#include <Puma/CTree.h>
```

Tree node representing a label statement. Derived from *CT_Statement*.

Example:

```
incr_a: a++;
```

10.1.3.4 CT_IfStmt

```
#include <Puma/CTree.h>
```

Tree node representing an if-statement. Derived from *CT_Statement* and *CSemScope*.

Example:

```
if (a==0) {
  a++;
}
```

10.1.3.5 CT_IfElseStmt

```
#include <Puma/CTree.h>
```

Tree node representing an if-else-statement. Derived from *CT_Statement* and *CSemScope*.

Example:

```
if (a==0) {
  a++;
} else {
  a=0;
}
```

10.1.3.6 CT_SwitchStmt

```
#include <Puma/CTree.h>
```

Tree node representing a switch statement. Derived from *CT_Statement* and *CSemScope*.

Example:

```
switch(a) {
  case 0: a++;
}
```

10.1.3.7 CT_BreakStmt

```
#include <Puma/CTree.h>
```

Tree node representing a break-statement. Derived from *CT_Statement*.

Example:

```
break;
```

10.1.3.8 CT_ExprStmt

```
#include <Puma/CTree.h>
```

Tree node representing an expression statement. Derived from *CT_Statement*.

Example:

```
a+b;
```

10.1.3.9 CT_WhileStmt

```
#include <Puma/CTree.h>
```

Tree node representing a while-statement. Derived from *CT_Statement* and *CSemScope*.

Example:

```
while (a>0) {
  a--;
}
```

10.1.3.10 CT_DoStmt

```
#include <Puma/CTree.h>
```

Tree node representing a do-while-statement. Derived from *CT_Statement*.

Example:

```
do {
  a--;
} while (a>0);
```

10.1.3.11 CT_ForStmt

```
#include <Puma/CTree.h>
```

Tree node representing a for-statement. Derived from *CT_Statement* and *CSemScope*.

Example:

```
for (int i=0; i<10; i++) {
  f(i);
}
```

10.1.3.12 CT_ContinueStmt

```
#include <Puma/CTree.h>
```

Tree node representing a continue-statement. Derived from *CT_Statement*.

Example:

```
continue;
```

10.1.3.13 CT_ReturnStmt

```
#include <Puma/CTree.h>
```

Tree node representing a return-statement. Derived from *CT_Statement*.

Example:

```
return 1;
```

10.1.3.14 CT_GotoStmt

```
#include <Puma/CTree.h>
```

Tree node representing a goto-stmt. Derived from *CT_Statement*.

Example:

```
goto incr_a;
```

10.1.3.15 CT_DeclStmt

```
#include <Puma/CTree.h>
```

Tree node representing a declaration statement. Derived from *CT_Statement*.

Example:

```
int i=0;
```

10.1.3.16 CT_CaseStmt

```
#include <Puma/CTree.h>
```

Tree node representing a case statement. Derived from *CT_Statement*.

Example:

```
case 42: a=42;
```

10.1.3.17 CT_DefaultStmt

```
#include <Puma/CTree.h>
```

Tree node representing a default statement of a switch statement. Derived from *CT_Statement*.

Example:

```
default: break;
```

10.1.3.18 CT_TryStmt

```
#include <Puma/CTree.h>
```

Tree node representing a try-catch statement. Derived from *CT_Statement*.

Example:

```
try {
  f();
} catch (...) {
  // call failed
}
```

10.1.4 Expressions

10.1.4.1 CT_Expression

```
#include <Puma/CTree.h>
```

Base class for all expression tree nodes. Derived from *CTree* and *CSemValue*.

10.1.4.2 CT_ExprList

```
#include <Puma/CTree.h>
```

Tree node representing an expression list. Derived from *CT_List*, *CSemValue*, and *CSemObject*.

10.1.4.3 CT_Call

```
#include <Puma/CTree.h>
```

Tree node representing explicit or implicit function calls including built-in or user-defined functions and overloaded operators. Derived from *CT_Expression* and *CSemObject*.

10.1.4.4 CT_CallExpr

```
#include <Puma/CTree.h>
```

Tree node representing a function call expression. Derived from *CT_Call*.

Example:

```
f(i)
```

10.1.4.5 CT_ImplicitCall

```
#include <Puma/CTree.h>
```

Tree node representing implicit function calls detected by the semantic analysis. Derived from *CT_Call*.

Example:

```
class Number {
  int _n;
public:
  Number(int n) : _n(n) {}
  int operator+(const Number& n) { return n._n + _n; }
};

Number one(1), two(2);
one + two;
// implicitely calls one.operator+(two)
```

10.1.4.6 CT_ThrowExpr

```
#include <Puma/CTree.h>
```

Tree node representing a throw expression. Derived from *CT_Expression*.

Example:

```
throw std::exception()
```

10.1.4.7 CT_NewExpr

```
#include <Puma/CTree.h>
```

Tree node representing a new expression. Derived from *CT_Expression* and *CSemObject*.

Example:

```
new A()
```

10.1.4.8 CT_DeleteExpr

```
#include <Puma/CTree.h>
```

Tree node representing a delete expression. Derived from *CT_Expression* and *CSemObject*.

Example:

```
delete a
```

10.1.4.9 CT_ConstructExpr

```
#include <Puma/CTree.h>
```

Tree node representing a construct expression. Derived from *CT_Expression* and *CSemObject*.

Example:

```
std::string("abc")
```

10.1.4.10 CT_Integer

```
#include <Puma/CTree.h>
```

Tree node representing an integer constant. Derived from *CT_Expression*.

Example:

```
1234
```

10.1.4.11 CT_Character

```
#include <Puma/CTree.h>
```

Tree node representing a single character constant. Derived from *CT_Expression*.

Example:

```
'a'
```

10.1.4.12 CT_WideCharacter

```
#include <Puma/CTree.h>
```

Tree node representing a wide character constant. Derived from *CT_Character*.

Example:

```
L'a'
```

10.1.4.13 CT_String

```
#include <Puma/CTree.h>
```

Tree node representing a string literal. Derived from *CT_List* and *CSemValue*.

Example:

```
"abc"
```

10.1.4.14 CT_WideString

```
#include <Puma/CTree.h>
```

Tree node representing a wide string literal. Derived from *CT_String*.

Example:

```
L"abc"
```

10.1.4.15 CT_Float

```
#include <Puma/CTree.h>
```

Tree node representing a floating point constant. Derived from *CT_Expression*.

Example:

```
12.34
```

10.1.4.16 CT_Bool

```
#include <Puma/CTree.h>
```

Tree node representing a boolean literal. Derived from *CT_Expression*.

Examples:

```
true
false
```

10.1.4.17 CT_BracedExpr

```
#include <Puma/CTree.h>
```

Tree node representing a braced expression. Derived from *CT_Expression*.

Examples:

```
(a+b)
```

10.1.4.18 CT_BinaryExpr

```
#include <Puma/CTree.h>
```

Tree node representing a binary expression. Derived from *CT_Call*.

Example:

```
a+b
```

10.1.4.19 CT_MembPtrExpr

```
#include <Puma/CTree.h>
```

Tree node representing a member pointer expression. Derived from *CT_Expression* and *CSemObject*.

Example:

```
a->b
```

10.1.4.20 CT_MembRefExpr

```
#include <Puma/CTree.h>
```

Tree node representing a member reference expression. Derived from *CT_MembPtrExpr*.

Example:

```
a.b
```

10.1.4.21 CT_UnaryExpr

```
#include <Puma/CTree.h>
```

Base class for tree nodes representing unary expressions. Derived from *CT_Call*.

Example:

```
!a
```

10.1.4.22 CT_PostfixExpr

```
#include <Puma/CTree.h>
```

Tree node representing a postfix expression. Derived from *CT_UnaryExpr*.

Example:

```
a++
```

10.1.4.23 CT_AddrExpr

```
#include <Puma/CTree.h>
```

Tree node representing an address expression. Derived from *CT_UnaryExpr*.

Example:

```
&a
```

10.1.4.24 CT_DerefExpr

`#include <Puma/CTree.h>`

Tree node representing a pointer dereferencing expression. Derived from *CT_UnaryExpr*.

Example:

```
*a
```

10.1.4.25 CT_IfThenExpr

`#include <Puma/CTree.h>`

Tree node representing an if-then expression. Derived from *CT_Expression*.

Examples:

```
a>0?a:b  // evaluate to a if a>0, and to b otherwise
a?:b     // short-cut for: a!=0?a:b
```

10.1.4.26 CT_CmpdLiteral

`#include <Puma/CTree.h>`

Tree node representing a compound literal. Derived from *CT_Expression* and *CSemObject*.

Example:

```
(int[]){1,2,3}
```

10.1.4.27 CT_IndexExpr

`#include <Puma/CTree.h>`

Tree node representing an index expression. Derived from *CT_Call*.

Example:

```
a[1]
```

10.1.4.28 CT_CastExpr

```
#include <Puma/CTree.h>
```

Tree node representing a cast expression. Derived from *CT_Expression*.

Example:

```
(int)a
```

10.1.4.29 CT_StaticCast

```
#include <Puma/CTree.h>
```

Tree node representing a static cast. Derived from *CT_Expression*.

Example:

```
static_cast<int>(a)
```

10.1.4.30 CT_ConstCast

```
#include <Puma/CTree.h>
```

Tree node representing a const cast. Derived from *CT_StaticCast*.

Example:

```
const_cast<int>(a)
```

10.1.4.31 CT_ReintCast

```
#include <Puma/CTree.h>
```

Tree node representing a reinterpret cast. Derived from *CT_StaticCast*.

Example:

```
reinterpret_cast<int>(a)
```

10.1.4.32 CT_DynamicCast

```
#include <Puma/CTree.h>
```

Tree node representing a dynamic cast. Derived from *CT_StaticCast*.

Example:

```
dynamic_cast<int>(a)
```

10.1.4.33 CT_TypeidExpr

```
#include <Puma/CTree.h>
```

Tree node representing a typeid expression. Derived from *CT_Expression*.

Example:

```
typeid(X)
```

10.1.4.34 CT_SizeofExpr

```
#include <Puma/CTree.h>
```

Tree node representing a sizeof expression. Derived from *CT_Expression*.

Example:

```
sizeof(int*)
```

10.1.4.35 CT_OffsetofExpr

```
#include <Puma/CTree.h>
```

Tree node representing an offsetof expression. Derived from *CT_Expression*.

Example:

```
offsetof(Circle,radius)
```

10.1.4.36 CT_ImplicitCast

```
#include <Puma/CTree.h>
```

Tree node representing an implicit cast. Derived from *CT_Expression*.

Example:

```
int i = 1.2; // implicit cast from float to int
```

10.1.4.37 CT_MembDesignator

```
#include <Puma/CTree.h>
```

Tree node representing a member designator. Derived from *CT_Expression*.

Example:

```
.a
```

10.1.4.38 CT_IndexDesignator

```
#include <Puma/CTree.h>
```

Tree node representing an index designator. Derived from *CT_Expression*.

Example:

```
[1]
```

10.1.4.39 CT_DesignatorSeq

```
#include <Puma/CTree.h>
```

Tree node representing a designator sequence. Derived from *CT_List* and *CSemValue*.

Example:

```
.a.b.c
```

10.1.5 Declaration Specifiers

10.1.5.1 CT_DeclSpec

```
#include <Puma/CTree.h>
```

Base class for all tree nodes representing declaration specifiers. Derived from *CTree*.

10.1.5.2 CT_DeclSpecSeq

```
#include <Puma/CTree.h>
```

Tree node representing a sequence of declaration specifiers. Derived from *CT_List*.

10.1.5.3 CT_PrimDeclSpec

```
#include <Puma/CTree.h>
```

Tree node representing a primitive declaration specifier. Derived from *CT_DeclSpec*.

Examples:

```
friend
extern
char
unsigned
```

10.1.5.4 CT_NamedType

```
#include <Puma/CTree.h>
```

Tree node representing a named type. Derived from *CT_DeclSpec* and *CSemObject*.

Example:

```
(int*)a
```

10.1.5.5 CT_ClassSpec

```
#include <Puma/CTree.h>
```

Tree node representing a class specifier. Derived from *CT_DeclSpec* and *CSemObject*.

Example:

```
class X
```

10.1.5.6 CT_UnionSpec

```
#include <Puma/CTree.h>
```

Tree node representing a union specifier. Derived from *CT_ClassSpec*.

Example:

```
union X
```

10.1.5.7 CT_EnumSpec

```
#include <Puma/CTree.h>
```

Tree node representing an enumeration specifier. Derived from *CT_ClassSpec*.

Example:

```
enum X
```

10.1.5.8 CT_ExceptionSpec

```
#include <Puma/CTree.h>
```

Tree node representing an exception specifier. Derived from CT_*DeclSpec*.

Example:

```
throw(std::exception)
```

10.1.5.9 CT_BaseSpec

```
#include <Puma/CTree.h>
```

Tree node representing a base class specifier. Derived from *CTree*.

Example:

```
public X
```

10.1.5.10 CT_BaseSpecList

```
#include <Puma/CTree.h>
```

Tree node representing a base specifier list. Derived from *CT_List*.

Example:

```
: public X, protected Y, Z
```

10.1.5.11 CT_AccessSpec

```
#include <Puma/CTree.h>
```

Tree node representing an access specifier. Derived from *CTree*.

Example:

```
public:
```

10.1.6 Declarators

10.1.6.1 CT_Declarator

```
#include <Puma/CTree.h>
```

Base class for all tree nodes representing declarators. Derived from *CTree*.

10.1.6.2 CT_DeclaratorList

```
#include <Puma/CTree.h>
```

Tree node representing a list of declarators. Derived from *CT_List*.

10.1.6.3 CT_InitDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing a declarator with initializer. Derived from *CT_Declarator* and *CSemObject*.

Example:

```
int *i = 0;
```

10.1.6.4 CT_BracedDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing a braced declarator. Derived from *CT_Declarator*.

Example:

```
int (i);
```

10.1.6.5 CT_ArrayDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing an array declarator. Derived from *CT_Declarator* and *CSemValue*.

Example:

```
a[10]
```

10.1.6.6 CT_ArrayDelimiter

```
#include <Puma/CTree.h>
```

Tree node representing an array delimiter. Derived from *CTree*.

Examples:

```
[10]
[*]
```

10.1.6.7 CT_FctDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing a function declarator. Derived from *CT_Declarator*.

Example:

```
f(int a) const
```

10.1.6.8 CT_RefDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing a reference declarator. Derived from *CT_Declarator*.

Example:

```
&a
```

10.1.6.9 CT_PtrDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing a pointer declarator. Derived from *CT_Declarator*.

Example:

```
*a
```

10.1.6.10 CT_MembPtrDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing a member pointer declarator. Derived from *CT_Declarator*.

Example:

```
*X::a
```

10.1.6.11 CT_BitFieldDeclarator

```
#include <Puma/CTree.h>
```

Tree node representing a bit-field declarator. Derived from *CT_Declarator* and *CSemObject*.

Example:

```
a : 2
```

10.1.7 Declarations

10.1.7.1 CT_Decl

```
#include <Puma/CTree.h>
```

Base class for all tree nodes representing declarations. Derived from *CTree*.

10.1.7.2 CT_DeclList

```
#include <Puma/CTree.h>
```

Tree node representing a list of declarations. Derived from *CT_List*.

10.1.7.3 CT_MembList

```
#include <Puma/CTree.h>
```

Tree node representing a member declarations list. Derived from *CT_DeclList* and *CSemScope*.

10.1.7.4 CT_ObjDecl

```
#include <Puma/CTree.h>
```

Tree node representing an object declaration. Derived from *CT_Decl*.

Example:

```
int *i
```

10.1.7.5 CT_ArgDecl

```
#include <Puma/CTree.h>
```

Tree node representing the declaration of a function parameter. Derived from *CT_Decl* and *CSemObject*.

10.1.7.6 CT_ArgDeclList

```
#include <Puma/CTree.h>
```

Tree node representing a function parameter list. Derived from *CT_DeclList* and *CSemScope*.

10.1.7.7 CT_ArgNameList

```
#include <Puma/CTree.h>
```

Tree node representing a K&R function parameter name list. Derived from *CT_ArgDeclList*.

10.1.7.8 CT_ArgDeclSeq

```
#include <Puma/CTree.h>
```

Tree node representing a K&R function parameter declarations list. Derived from *CT_DeclList* and *CSemScope*.

10.1.7.9 CT_AccessDecl

```
#include <Puma/CTree.h>
```

Tree node representing a member access declaration. Derived from *CT_Decl*.

Example:

```
baseClassMember;
```

10.1.7.10 CT_UsingDecl

```
#include <Puma/CTree.h>
```

Tree node representing a using declaration. Derived from *CT_AccessDecl*.

Example:

```
using Base::m_Member;
```

10.1.7.11 CT_AsmDef

```
#include <Puma/CTree.h>
```

Tree node representing an inline assembly definition. Derived from *CT_Decl*.

Example:

```
asm("movl %ecx %eax");
```

10.1.7.12 CT_EnumDef

```
#include <Puma/CTree.h>
```

Tree node representing the definition of an enumeration. Derived from *CT_Decl* and *CSemObject*.

Example:

```
enum E { A, B, C }
```

10.1.7.13 CT_ClassDef

```
#include <Puma/CTree.h>
```

Tree node representing a class definition. Derived from *CT_Decl* and *CSemObject*.

Example:

```
class X : Y { int x; }
```

10.1.7.14 CT_UnionDef

```
#include <Puma/CTree.h>
```

Tree node representing the definition of a union. Derived from *CT_ClassDef*.

Example:

```
union U { int i; }
```

10.1.7.15 CT_Enumerator

```
#include <Puma/CTree.h>
```

Tree node representing a single enumeration constant. Derived from *CT_Decl* and *CSemObject*.

10.1.7.16 CT_EnumeratorList

```
#include <Puma/CTree.h>
```

Tree node representing a list of enumerator constants. Derived from *CT_List*.

10.1.7.17 CT_LinkageSpec

```
#include <Puma/CTree.h>
```

Tree node representing a list of declaration with a specific linkage. Derived from *CT_Decl*.

10.1.7.18 CT_Handler

```
#include <Puma/CTree.h>
```

Tree node representing an exception handler. Derived from *CT_Decl* and *CSemScope*.

10.1.7.19 CT_TemplateDecl

```
#include <Puma/CTree.h>
```

Tree node representing a template declaration. Derived from *CT_Decl* and *CSemScope*.

10.1.7.20 CT_TemplateParamDecl

```
#include <Puma/CTree.h>
```

Base class for all tree nodesrepresenting a template parameter declaration. Derived from *CT_Decl* and *CSemObject*.

10.1.7.21 CT_TypeParamDecl

```
#include <Puma/CTree.h>
```

Tree node representing a template type parameter declaration. Derived from *CT_TemplateParamDecl*.

10.1.7.22 CT_NonTypeParamDecl

```
#include <Puma/CTree.h>
```

Tree node representing a template non-type parameter declaration. Derived from *CT_TemplateParamDecl*.

10.1.7.23 CT_TemplateParamList

```
#include <Puma/CTree.h>
```

Tree node representing a template parameter list. Derived from *CT_List* and *CSemScope*.

10.1.7.24 CT_TemplateArgList

```
#include <Puma/CTree.h>
```

Tree node representing a template argument list. Derived from *CT_List*.

10.1.7.25 CT_NamespaceDef

```
#include <Puma/CTree.h>
```

Tree node representing a namespace definition. Derived from *CT_Decl* and *CSemObject*.

Example:

```
namespace a {}
```

10.1.7.26 CT_NamespaceAliasDef

```
#include <Puma/CTree.h>
```

Tree node representing a namespace alias definition. Derived from *CT_Decl* and *CSemObject*.

Example:

```
namespace b = a;
```

10.1.7.27 CT_UsingDirective

```
#include <Puma/CTree.h>
```

Tree node representing a namespace using directive. Derived from *CT_Decl*.

Example:

```
using namespace std;
```

10.1.7.28 CT_Condition

```
#include <Puma/CTree.h>
```

Tree node representing a control-statement condition. Derived from *CT_Decl* and *CSemObject*.

Example:

```
int i = 0
```

10.1.7.29 CT_FctDef

```
#include <Puma/CTree.h>
```

Tree node representing a function definition. Derived from *CT_Decl* and *CSemObject*.

Example:

```
int mul(int x, int y) {
  return x*y;
}
```

10.1.7.30 CT_MembInitList

```
#include <Puma/CTree.h>
```

Tree node representing a constructor initializer list. Derived from *CT_List* and *CSemScope*.

Example:

```
: BaseClass(), m_Member(0)
```

10.1.7.31 CT_HandlerSeq

```
#include <Puma/CTree.h>
```

Tree node representing an exception handler sequence. Derived from *CT_List*.

10.1.8 Names

10.1.8.1 CT_SimpleName

```
#include <Puma/CTree.h>
```

Base class for all tree nodes representing a name. Derived from *CT_List*, *CSemValue*, *CSemObject*, and *Printable*.

Example:

```
a
```

10.1.8.2 CT_SpecialName

```
#include <Puma/CTree.h>
```

Base class for tree nodes representing a special name, like destructor names. Derived from *CT_SimpleName*.

10.1.8.3 CT_PrivateName

```
#include <Puma/CTree.h>
```

Tree node representing a private name. Derived from *CT_SpecialName*.

Private names are generated names for instance for abstract declarators.

Example:

```
void foo(int*);
// first parameter of foo has a private name
```

10.1.8.4 CT_OperatorName

```
#include <Puma/CTree.h>
```

Tree node representing the name of an overloaded operator. Derived from *CT_SpecialName*.

Example:

```
operator==
```

10.1.8.5 CT_DestructorName

```
#include <Puma/CTree.h>
```

Tree node representing a destructor name. Derived from *CT_SpecialName*.

Example:

```
~X
```

10.1.8.6 CT_ConversionName

```
#include <Puma/CTree.h>
```

Tree node representing the name of a conversion function. Derived from *CT_SpecialName*.

Example:

```
operator int*
```

10.1.8.7 CT_TemplateName

```
#include <Puma/CTree.h>
```

Tree node representing a template name. Derived from *CT_SpecialName*.

Example:

```
X<T>
```

10.1.8.8 CT_QualName

```
#include <Puma/CTree.h>
```

Tree node representing a qualified name. Derived from *CT_SimpleName*.

Example:

```
X::Y::Z
```

10.1.8.9 CT_RootQualName

```
#include <Puma/CTree.h>
```

Tree node representing a qualified name with introducing name separator. Derived from *CT_QualName*.

Example:

```
::X::Y::Z
```

10.1.9 Wildcards

10.1.9.1 CT_Any

```
#include <Puma/CTree.h>
```

Tree node representing a wildcard. Derived from *CTree*.

10.1.9.2 CT_AnyList

```
#include <Puma/CTree.h>
```

Tree node representing a list wildcard. Derived from *CT_Any*.

10.1.9.3 CT_AnyExtension

```
#include <Puma/CTree.h>
```

Tree node representing a wildcard extension. Derived from *CTree* and *CSemValue*.

10.1.9.4 CT_AnyCondition

```
#include <Puma/CTree.h>
```

Tree node representing the condition of a wildcard. Derived from *CTree*.

10.1.10 AspectC++

10.1.10.1 CT_AdviceDecl

```
#include <Puma/ACTree.h>
```

Tree node representing an advice declaration. Derived from *CT_Decl*.

Example:

```
advice "% main(...)" : before() {
  printf('init');
}
```

10.1.10.2 CT_OrderList

```
#include <Puma/ACTree.h>
```

Tree node representing a pointcut order list. Derived from *CT_List*.

Example:

```
( "pointcut1", "pointcut2" )
```

10.1.10.3 CT_PointcutDecl

```
#include <Puma/ACTree.h>
```

Tree node representing a pointcut declaration. Derived from *CT_Decl*.

Example:

```
pointcut main() = "% main(...)";
```

10.1.10.4 CT_Intro

```
#include <Puma/ACTree.h>
```

Tree node representing an introduction advice declaration. Derived from *CT_List* and *CSemScope*.

Example:

```
around()
```

10.1.10.5 CT_ClassSliceDecl

```
#include <Puma/ACTree.h>
```

Tree node representing a slice declaration for a class. Derived from *CTree* and *CSemObject*.

Example:

```
slice class X : Y {
  int x;
};
```

10.1.10.6 CT_SliceRef

```
#include <Puma/ACTree.h>
```

Tree node representing a slice reference. Derived from *CTree*.

Example:

```
slice X;
```

10.1.11 VisualC++

10.1.11.1 CT_AsmBlock

```
#include <Puma/WinCTree.h>
```

Tree node representing an inline assembly block. Derived from *CT_Statement*.

Example:

```
asm { movl ecx eax }
```

10.1.12 GNU C/C++

10.1.12.1 CT_GnuAsmSpec

```
#include <Puma/GnuCTree.h>
```

Tree node representing an extended inline assembly specifier. Derived from *CTree*.

Example:

```
asm("r0")
```

10.1.12.2 CT_GnuAsmDef

```
#include <Puma/GnuCTree.h>
```

Tree node representing an extended inline assembly definition. Derived from *CT_AsmDef*.

Example:

```
asm("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

10.1.12.3 CT_GnuAsmOperand

```
#include <Puma/GnuCTree.h>
```

Tree node representing an extended inline assembly operand. Derived from *CTree*.

Example:

```
"=f" (result)
```

10.1.12.4 CT_GnuAsmOperands

#include <Puma/GnuCTree.h>

Tree node representing a list of extended inline assembly operands. Derived from *CT_List*.

Example:

```
: "=f" (result) : "f" (angle)
```

10.1.12.5 CT_GnuAsmClobbers

#include <Puma/GnuCTree.h>

Tree node representing a list of extended inline assembly clobbers. Derived from *CT_List*.

Example:

```
: "r1", "r2", "r3", "r4", "r5"
```

10.1.12.6 CT_GnuStatementExpr

#include <Puma/GnuCTree.h>

Tree node representing a statement expression. Derived from *CT_Expression*.

Example:

```
({ int i = 0; i++; })
```

10.1.12.7 CT_GnuTypeof

#include <Puma/GnuCTree.h>

Tree node representing a typeof expression. Derived from *CT_DeclSpec* and *CSemValue*.

Example:

```
typeof(a+b)
```

10.2 Semantic Tree Classes

10.2.1 Basic Semantic Classes

10.2.1.1 CObjectInfo

```
#include <Puma/CObjectInfo.h>
```

Abstract base class of all semantic information classes.

Provides all semantic information about an entity (class, function, object, etc).

A semantic object is identified by its object ID. Semantic information objects for the same kind of entity have the same object ID (like object ID *CObjectInfo::FUNCTION_INFO* for all semantic objects of functions).

Example:

```
// check if sem_obj is a semantic object for a function
if (sem_obj.Id() == Puma::CObjectInfo::FUNCTION_INFO) {
  ...
}
// same check
if (sem_obj.FunctionInfo()) {
  ...
}
```

Semantic information objects are created by the semantic analysis component of Puma (see class *Semantic*) during the parse process and are collected in the semantic information database (see class *CSemDatabase*).

There are several relations between the semantic objects forming the semantic tree. There is one semantic tree for each translation unit.

The root of the semantic tree usually is the semantic object for the file scope (see class *CFileInfo*). It contains all the other scopes of the analysed source file, such as namespaces and class definitions, function definitions, global variables, and so on. The semantic tree is destroyed by destroying the root object of the tree. This recursively destroys all sub-objects of the tree.

10.2.1.2 CLanguage

```
#include <Puma/CLanguage.h>
```

Language specific encoding of entity names.

The language is specified using the 'extern' linkage specifier.

Following languages are supported.

Language Type Constants	Language
CLanguage::LANG_C	Language C.
CLanguage::LANG_CPLUSPLUS	Language C++.
CLanguage::LANG_OTHER	Neither C nor C++.
CLanguage::LANG_UNDEFINED	No explicit language encoding.

C entity names are not encoded. C++ entity names are encoded according to the *C++ V3 ABI mangling*.

Example:

```
void foo(char);          // encoded as: _Z3fooc
extern "C" void bar(int);  // encoded as: bar
```

10.2.1.3 CSpecifiers

```
#include <Puma/CSpecifiers.h>
```

C/C++ declaration specifiers for the declaration of an entity. The following declaration specifiers are supported.

Specifier Constant	Represented Specifier
CSpecifiers::SPEC_VIRTUAL	*virtual*
CSpecifiers::SPEC_STATIC	*static*
CSpecifiers::SPEC_EXTERN	*extern*
CSpecifiers::SPEC_MUTABLE	*mutable*
CSpecifiers::SPEC_REGISTER	*register*
CSpecifiers::SPEC_EXPLICIT	*explicit*

CSpecifiers::SPEC_AUTO	*auto*
CSpecifiers::SPEC_INLINE	*inline*
CSpecifiers::SPEC_NONE	No declaration specifier.

10.2.1.4 CLinkage

```
#include <Puma/CLinkage.h>
```

Linkage of an entity name (object, function, etc).

The linkage controls where a name is visible. There are three types of linkage: internal, external, and no linkage.

Linkage Type Constant	Linkage
CLinkage::LINK_INTERNAL	Internal linkage.
CLinkage::LINK_EXTERNAL	External linkage.
CLinkage::LINK_NONE	No linkage.

Names with external linkage are visible outside the object file where they occur. Names with internal or no linkage are only visible in one object file.

The linkage is implicitely defined by the scope in which the entity is declared. With the linkage specifier *extern* an entity name can be explicitely declared to have external linkage.

10.2.1.5 CProtection

```
#include <Puma/CProtection.h>
```

Access protection of C++ class members for the purpose of member access control.

There are three kinds of protection: private, public, and protected.

Protection Type Constant	Meaning
CProtection::PROT_PUBLIC	Public member access.
CProtection::PROT_PROTECTED	Protected member access.
CProtection::PROT_PRIVATE	Private member access.
CProtection::PROT_NONE	Undefined member access.

The protection either is defined implicitely or explicitely using member access specifiers.

10.2.1.6 CStorage

```
#include <Puma/CStorage.h>
```

Storage class of an object.

Defines the minimum potential lifetime of the storage containing an object. There are three different storage classes: static, automatic, and dynamic.

Storage Class Constant	**Meaning**
CProtection::CLASS_STATIC	Static storage class.
CProtection::CLASS_AUTOMATIC	Automatic storage class.
CProtection::CLASS_DYNAMIC	Dynamic storage class.
CProtection::CLASS_NONE	Undefined storage class.

10.2.2 C/C++

10.2.2.1 CFileInfo

```
#include <Puma/CFileInfo.h>
```

Semantic information about a source file (translation unit). Derived from *CNamespaceInfo*.

A source file has its own scope, the so-called file scope.

10.2.2.2 CEnumeratorInfo

```
#include <Puma/CEnumeratorInfo.h>
```

Semantic information about an enumeration constant. Derived from *CAttributeInfo*.

An enumeration constant also is called enumerator.

10.2.2.3 CUsingInfo

```
#include <Puma/CUsingInfo.h>
```

Semantic information about a using-directive. Derived from *CScopeRequest*.

The using-directive makes names from a namespace visible in another namespace or scope.

Example:

```
namespace A {
  class X {};
}
using namespace A;  // make A::X visible in global scope

X x;   // resolves to A::X
```

10.2.2.4 CUnionInfo

```
#include <Puma/CUnionInfo.h>
```

Semantic information about a union. Derived from *CRecord*.

10.2.2.5 CNamespaceInfo

```
#include <Puma/CNamespaceInfo.h>
```

Semantic information about a user-defined namespace. Derived from *CStructure*.

There are two kinds of namespaces: original namespaces and namespace aliases.

10.2.2.6 CSourceInfo

```
#include <Puma/CSourceInfo.h>
```

Source file information for an entity.

Stores the file information and start token of the entity in the source file.

10.2.2.7 CRecord

```
#include <Puma/CRecord.h>
```

Semantic information about a class or union. Derived from *CStructure*.

10.2.2.8 CArgumentInfo

```
#include <Puma/CArgumentInfo.h>
```

Semantic information about a function parameter. Derived from *CScopeRequest*.

10.2.2.9 CTemplateInstance

```
#include <Puma/CTemplateInstance.h>
```

Semantic information about a template instance.

Contains the point of instantiation, the instantiated template, the instantiation arguments, and the deduced template arguments.

The point of instantiation (POI) is the corresponding template-id.

```
X<int> x;   // X<int> is the POI
```

The instantiation arguments are the arguments of the template-id at the POI.

```
Y<int,1> y; // 'int' and '1' are the instantiation
   arguments
```

The deduced template arguments are calculated from the instantiation arguments and the template default arguments.

```
template<class T1, class T2 = float>
struct Foo {
  template<class T3, class T4>
  T1 foo(T2,T3,T4*);
};

void bar(bool b, char* s) {
  Foo<int> f;    // deduced arguments: T1=int, T2=float
  f.foo(1,b,s); // deduced arguments: T3=bool, T4=char
}
```

If a template instance is not yet created (maybe because real template instantiation is disabled or due to late template instantiation), then this template instance is called a pseudo instance.

10.2.2.10 CFctInstance

```
#include <Puma/CFctInstance.h>
```

Semantic information about an instance of a function template. Derived from *CFunctionInfo*.

10.2.2.11 CFunctionInfo

```
#include <Puma/CFunctionInfo.h>
```

Semantic information about a function, method, overloaded operator, or user conversion function. Derived from *CStructure*.

10.2.2.12 CScopeRequest

```
#include <Puma/CScopeRequest.h>
```

Provides additional scope information for semantic objects that do not represent scopes itself (like objects). Derived from *CObjectInfo*.

10.2.2.13 CScopeInfo

```
#include <Puma/CScopeInfo.h>
```

Semantic information about a scope. Derived from *CObjectInfo*.

Several syntactic constructs have its own scope, such as class definitions, functions, and compound statements.

10.2.2.14 CClassInstance

```
#include <Puma/CClassInstance.h>
```

Semantic information about an instance of a class template. Derived from *CClassInfo*.

10.2.2.15 CLabelInfo

```
#include <Puma/CLabelInfo.h>
```

Semantic information about a jump label. Derived from *CScopeRequest*.

Jump labels are used as argument of goto-statements.

10.2.2.16 CTypedefInfo

```
#include <Puma/CTypedefInfo.h>
```

Semantic information about a typedef. Derived from *CScopeRequest*.

A typedef is a named type for any underlying type. The type of a typedef is the underlying type.

10.2.2.17 CUnionInstance

```
#include <Puma/CUnionInstance.h>
```

Semantic information about an instance of a union template. Derived from *CUnionInfo*.

10.2.2.18 CAttributeInfo

```
#include <Puma/CAttributeInfo.h>
```

Semantic information about a local or global object or a class data member. Derived from *CScopeRequest*.

10.2.2.19 CEnumInfo

```
#include <Puma/CEnumInfo.h>
```

Semantic information about an enumeration. Derived from *CScopeRequest*.

10.2.2.20 CTemplateInfo

```
#include <Puma/CTemplateInfo.h>
```

Semantic information about a template declaration. Derived from *CStructure*.

Contains information about the parameters, specializations, and instances of a template.

10.2.2.21 CMemberAliasInfo

`#include <Puma/CMemberAliasInfo.h>`

Semantic information about a member alias. Derived from *CScopeRequest*.

A member alias is created by a using-declaration.

10.2.2.22 CStructure

`#include <Puma/CStructure.h>`

Base class of all semantic information classes for entities that can contain other entity declarations (like classes, namespaces, functions). Derived from *CScopeInfo*.

10.2.2.23 CTemplateParamInfo

`#include <Puma/CTemplateParamInfo.h>`

Semantic information about a template parameter. Derived from *CObjectInfo*.

There are three kinds of template parameters: type, non-type, and template template parameters.

```
// T is a type template parameter
// I is a non-type template parameter
// TT is a template template parameter
template<class T, int I, template<typename,int> class TT>
class X {
  TT<T,I> x;
};
```

10.2.2.24 CBaseClassInfo

`#include <Puma/CBaseClassInfo.h>`

Semantic information about a base class of a class. Derived from *CScopeRequest*.

The base class is specified by a base class specifier in the base class list of a class definition. It can have several qualifiers like *virtual*, *public*, and so on.

10.2.2.25 CClassInfo

```
#include <Puma/CClassInfo.h>
```

Semantic information about a class. Derived from *CRecord*.

Note that *struct*s are ordinary classes where the member access type defaults to *public*. Note also that a union, although syntactically very similar, is not a class and thus not represented by *CClassInfo*.

10.2.2.26 CLocalScope

```
#include <Puma/CLocalScope.h>
```

Semantic information about a local scope in a function body, also called block scope. Derived from *CStructure*.

Examples for local scopes:

```
{ <local scope> }
if (...) <local scope>;
while (...) { <local scope> }
```

10.2.3 AspectC++

10.2.3.1 ACAdviceInfo

```
#include <Puma/ACAdviceInfo.h>
```

Semantic information about an AspectC++ *advice* declaration.

An advice is part of an aspect declaration.

10.2.3.2 ACAspectInfo

```
#include <Puma/ACAspectInfo.h>
```

Semantic information about an AspectC++ *aspect* declaration.

An aspect declaration is syntactically equal to a C++ class declaration and also parsed like a C++ class. Additionally it contains *pointcut* and *advice* declarations.

10.2.3.3 ACSliceInfo

```
#include <Puma/ACSliceInfo.h>
```

Semantic information about an AspectC++ *slice* declaration.

A slice represents a fragment of a C/C++ language element. For example a *class slice* is a (possibly incomplete) fragment of a class.

10.2.3.4 ACIntroductionInfo

```
#include <Puma/ACIntroductionInfo.h>
```

Semantic information about an AspectC++ *introduction advice* declaration.

10.2.3.5 ACPointcutInfo

```
#include <Puma/ACPointcutInfo.h>
```

Semantic information about an AspectC++ *pointcut* declaration.

A pointcut declaration is syntactically equal to a function declaration and also parsed like a function.

10.3 Type Information Classes

10.3.1 CTypeInfo

```
#include <Puma/CTypeInfo.h>
```

Type information for an entity (class, function, object, etc).

There are two kinds of types: fundamental types like *int*, and compound types like *int**. Types describe objects, references, or functions.

A type is identified by its ID.

```
// check if type is a function type
if (type.Id() == Puma::CTypeInfo::TYPE_FUNCTION) {
  ...
}
// same check
if (type.TypeFunction()) {
  ...
}
// same check
if (type.isFunction()) {
  ...
}
```

10.3.2 CTypeList

```
#include <Puma/CTypeList.h>
```

List of types. Used for instance for the list of function parameter types.

10.3.3 CTypeAddress

```
#include <Puma/CTypeInfo.h>
```

Type of a reference.

Examples:

```
int& i = i0;
// i has type 'reference to int'
// type structure:
// CTypeAddress
//   CTypePrimitive int

const X& x = x0;
// x has type 'reference to const X'
// type structure:
// CTypeAddress
//   CTypeQualified const
//     CTypeClass X
```

10.3.4 CTypeVarArray

`#include <Puma/CTypeInfo.h>`

Type of a variable length array.

Example:

```
void foo(int len) {
  int i[len];
  // i has type 'variable length array of int'
}
```

10.3.5 CTypeUnion

`#include <Puma/CTypeInfo.h>`

Type of a union.

Example:

```
union X x;
// x has type 'union X'
```

10.3.6 CTypeEnum

```
#include <Puma/CTypeInfo.h>
```

Type of an enumeration.

Examples:

```
enum E { A,B } e;
// e has type 'enum E'

enum { C,D } a;
// a has type 'enum <anonymous>'
```

10.3.7 CTypeBitField

```
#include <Puma/CTypeInfo.h>
```

Type of a bit-field.

Example:

```
class X {
  int i : 10;
  // i has type 'bit-field of size 10'
  // type structure:
  // CTypeBitField dim=10
  //   CTypePrimitive int
};
```

10.3.8 CTypeQualified

```
#include <Puma/CTypeInfo.h>
```

Type qualification. There are three type qualifier: *const*, *volatile*, and *restrict*.

Examples:

```
const int i = 0;
// i has type 'const int'
// type structure:
// CTypeQualified const
//   CTypePrimitive int

char * const s = 0;
// s has type 'const pointer to char'
// type structure:
// CTypeQualified const
//   CTypePointer
//     CTypePrimitive char
```

10.3.9 CTypeMemberPointer

```
#include <Puma/CTypeInfo.h>
```

Type of a member pointer.

Examples:

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

int X::* aptr = &X::a;
// aptr has type 'member pointer to int'
// type structure:
// CTypeMemberPointer class=X
//   CTypePrimitive int

void (X::*fptr)(int) = &X::f;
// fptr has type 'member pointer to function
// returning void with one argument int'
// type structure:
// CTypeMemberPointer class=X
```

```
//   CTypeFunction args=int
//     CTypePrimitive void
```

10.3.10 CTypeFunction

```
#include <Puma/CTypeInfo.h>
```

Type of a function.

Example:

```
void foo(int);
// foo has type 'function returning void
// with one argument int'
// type structure:
// CTypeFunction args=int
//   CTypePrimitive void
```

10.3.11 CTypePointer

```
#include <Puma/CTypeInfo.h>
```

Type of a pointer.

Examples:

```
int* ip = 0;
// ip has type 'pointer to int'
// type structure:
// CTypePointer
//   CTypePrimitive int

const char* s = 0;
// s has type 'pointer to const char'
// type structure:
// CTypePointer
//   CTypeQualified const
//     CTypePrimitive char
```

10.3.12 CTypeRecord

```
#include <Puma/CTypeInfo.h>
```

Type of a class or union.

10.3.13 CTypeArray

```
#include <Puma/CTypeInfo.h>
```

Type of an array.

Examples:

```
int i[10];
// i has type 'array of int'
// type structure:
// CTypeArray dim=10
//   CTypePrimitive int

char* sa[5];
// sa has type 'array of pointer to char'
// type structure:
// CTypeArray dim=5
//   CTypePointer
//     CTypePrimitive char
```

10.3.14 CTypePrimitive

```
#include <Puma/CTypeInfo.h>
```

Primitive type. The fundamental arithmetic types and type *void* are called primitive types.

Following primitve types are defined.

Predefined Type Object	Represented Type
Puma::CTYPE_BOOL	*bool*
Puma::CTYPE_C_BOOL	*_Bool*
Puma::CTYPE_CHAR	*char*
Puma::CTYPE_SIGNED_CHAR	*signed char*
Puma::CTYPE_UNSIGNED_CHAR	*unsigned char*
Puma::CTYPE_WCHAR_T	*wchar_t*
Puma::CTYPE_SHORT	*short*
Puma::CTYPE_UNSIGNED_SHORT	*unsigned short*
Puma::CTYPE_INT	*int*
Puma::CTYPE_UNSIGNED_INT	*unsigned int*
Puma::CTYPE_LONG	*long*
Puma::CTYPE_UNSIGNED_LONG	*unsigned long*
Puma::CTYPE_LONG_LONG	*long long*
Puma::CTYPE_UNSIGNED_LONG_LONG	*unsigned long long*
Puma::CTYPE_FLOAT	*float*
Puma::CTYPE_DOUBLE	*double*
Puma::CTYPE_LONG_DOUBLE	*long double*
Puma::CTYPE_VOID	*void*
Puma::CTYPE_UNKNOWN_T	*unknown_t*
Puma::CTYPE_UNDEFINED	Undefined type.
Puma::CTYPE_ELLIPSIS	Any type.

10.3.15 CTypeClass

```
#include <Puma/CTypeInfo.h>
```

Type of a class.

Examples:

```
class X x;
// x has type 'class X'
```

```
struct Y y;
// y has type 'class Y'
```

10.3.16 CTypeTemplateParam

```
#include <Puma/CTypeInfo.h>
```

Type of a template parameter.

10.4 Preprocessor Syntax Tree Classes

10.4.1 PreTree

```
#include <Puma/PreTree.h>
```

Base class for all C preprocessor syntax tree nodes.

10.4.2 PreTreeComposite

```
#include <Puma/PreTreeComposite.h>
```

Base class for all C preprocessor syntax tree composite nodes. Derived from *PreTree*.

10.4.3 PreProgram

```
#include <Puma/PreTreeNodes.h>
```

The root node of the preprocessor syntax tree. Derived from *PreTreeComposite*.

10.4.4 PreDirectiveGroups

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing the directive groups in the program. Derived from *PreTreeComposite*.

10.4.5 PreConditionalGroup

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a group of conditional directives. Derived from *PreTreeComposite*.

Example:

```
#if
...
#elif
...
#else
...
#endif
```

10.4.6 PreElsePart

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a group of directives in the #else part of an #if conditional. Derived from *PreTreeComposite*.

10.4.7 PreElifPart

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a group of directives in the #elif part of an #if conditional. Derived from *PreTreeComposite*.

10.4.8 PreIfDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #if directive. Derived from *PreTreeComposite*.

Example:

```
#if OSTYPE==Linux
```

10.4.9 PreIfdefDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #ifdef directive. Derived from *PreTreeComposite*.

Example:

```
#ifdef Linux
```

10.4.10 PreIfndefDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #ifndef directive. Derived from *PreTreeComposite*.

Example:

```
#ifndef Linux
```

10.4.11 PreElifDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #elif directive. Derived from *PreTreeComposite*.

Example:

```
#elif OSTYPE==linux
```

10.4.12 PreElseDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #else directive. Derived from *PreTreeComposite*.

Example:

```
#else
```

10.4.13 PreEndifDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #endif directive. Derived from *PreTreeComposite*.

Example:

```
#endif
```

10.4.14 PreIncludeDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #include or #include_next directive. Derived from *PreTreeComposite*.

Example:

```
#include <stdio.h>
#include_next <stdio.h>
```

10.4.15 PreAssertDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #assert directive. Derived from *PreTreeComposite*.

Example:

```
#assert OSTYPE (linux)
```

10.4.16 PreUnassertDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #unassert directive. Derived from *PreTreeComposite*.

Example:

```
#unassert OSTYPE
```

10.4.17 PreDefineFunctionDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a #define directive for function-like macros. Derived from *PreTreeComposite*.

Example:

```
#define MUL(a,b) (a * b)
```

10.4.18 PreDefineConstantDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a #define directive for constants. Derived from *PreTreeComposite*.

Example:

```
#define CONSTANT 1
```

10.4.19 PreUndefDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #undef directive. Derived from *PreTreeComposite*.

Example:

```
#undef MACRO
```

10.4.20 PreWarningDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a #warning directive. Derived from *PreTreeComposite*.

Example:

```
#warning This is a warning.
```

10.4.21 PreErrorDirective

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing an #error directive. Derived from *PreTreeComposite*.

Example:

```
#error This is an error.
```

10.4.22 PreIdentifierList

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing the identifier list of a function-like macro definition. Derived from *PreTreeComposite*.

Example:

```
a,b,c
```

10.4.23 PreTokenList

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing the token list of a macro body. Derived from *PreTreeComposite*.

10.4.24 PreTokenListPart

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a part of the token list of a macro body. Derived from *PreTreeComposite*.

10.4.25 PreCondSemNode

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor semantic tree node for conditions. Derived from *PreTree*.

10.4.26 PreInclSemNode

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor semantic tree node for the #include directive containing the unit to include. Derived from *PreTree*.

10.4.27 PreError

```
#include <Puma/PreTreeNodes.h>
```

Preprocessor tree node representing a parse error. Derived from *PreTree*.

10.5 Tokens

10.5.1 C/C++ Tokens

```
#include <Puma/CTokens.h>
```

Token Type Constant	Represented Text
Puma::TOK_AT	`@`
Puma::TOK_ADD_EQ	`&&`
Puma::TOK_ADVICE	`advice`
Puma::TOK_AND	`&`
Puma::TOK_AND_AND	`&&`
Puma::TOK_AND_AND_ISO_646	`and`
Puma::TOK_AND_EQ	`&=`
Puma::TOK_AND_EQ_ISO_646	`and_eq`
Puma::TOK_AND_ISO_646	`bitand`
Puma::TOK_ASM	`asm`
Puma::TOK_ASM_2	`__asm`
Puma::TOK_ASM_3	`__asm__`
Puma::TOK_ASPECT	`aspect`
Puma::TOK_ASSIGN	`=`
Puma::TOK_AUTO	`auto`
Puma::TOK_BOOL	`bool`
Puma::TOK_BOOL_VAL	`true, false`
Puma::TOK_BREAK	`break`
Puma::TOK_C_BOOL	`_Bool`
Puma::TOK_CASE	`case`
Puma::TOK_CATCH	`catch`
Puma::TOK_CDECL	`_cdecl`
Puma::TOK_CDECL_2	`__cdecl`
Puma::TOK_CHAR	`char`
Puma::TOK_CHAR_VAL	Character constant like `'a'` or `L'a'`.
Puma::TOK_CLASS	`class`
Puma::TOK_CLOSE_CURLY	`}`
Puma::TOK_CLOSE_ROUND	`)`

Token Type Constant	Represented Text
Puma::TOK_CLOSE_SQUARE	`]`
Puma::TOK_COLON	`:`
Puma::TOK_COLON_COLON	`::`
Puma::TOK_COMMA	`,`
Puma::TOK_CONST	`const`
Puma::TOK_CONST_2	`__const`
Puma::TOK_CONST_3	`__const__`
Puma::TOK_CONST_CAST	`const_cast`
Puma::TOK_CONTINUE	`continue`
Puma::TOK_DECR	`--`
Puma::TOK_DEFAULT	`default`
Puma::TOK_DELETE	`delete`
Puma::TOK_DIV	`/`
Puma::TOK_DIV_EQ	`/=`
Puma::TOK_DO	`do`
Puma::TOK_DOT	`.`
Puma::TOK_DOT_STAR	`.*`
Puma::TOK_DOUBLE	`double`
Puma::TOK_DYN_CAST	`dynamic_cast`
Puma::TOK_ELLIPSIS	`...`
Puma::TOK_ELSE	`else`
Puma::TOK_ENUM	`enum`
Puma::TOK_EQL	`==`
Puma::TOK_EXPLICIT	`explicit`
Puma::TOK_EXPORT	`export`
Puma::TOK_EXTERN	`extern`
Puma::TOK_FASTCALL	`_fastcall`
Puma::TOK_FASTCALL_2	`__fastcall`
Puma::TOK_FLOAT	`float`
Puma::TOK_FLT_VAL	Floating point constant like `12.34`.
Puma::TOK_FOR	`for`
Puma::TOK_FRIEND	`friend`
Puma::TOK_GEQ	`>=`
Puma::TOK_GOTO	`goto`

Token Type Constant	**Represented Text**
Puma::TOK_GREATER	`>`
Puma::TOK_ID	Any identifier that is not a keyword.
Puma::TOK_IF	`if`
Puma::TOK_IF_EXISTS	`__if_exists`
Puma::TOK_IF_NOT_EXISTS	`__if_not_exists`
Puma::TOK_INCR	`++`
Puma::TOK_INLINE	`inline`
Puma::TOK_INLINE_2	`__inline`
Puma::TOK_INLINE_3	`__inline__`
Puma::TOK_INT	`int`
Puma::TOK_INT_VAL	Integer constant like `1234`.
Puma::TOK_INT64	`__int64`
Puma::TOK_IOR_EQ	`|=`
Puma::TOK_IOR_EQ_ISO_646	`or_eq`
Puma::TOK_LEQ	`<=`
Puma::TOK_LESS	`<`
Puma::TOK_LONG	`long`
Puma::TOK_LSH	`<<`
Puma::TOK_LSH_EQ	`<<=`
Puma::TOK_MINUS	`-`
Puma::TOK_MOD_EQ	`=`
Puma::TOK_MODULO	`%`
Puma::TOK_MUL	`*`
Puma::TOK_MUL_EQ	`*=`
Puma::TOK_MUTABLE	`mutable`
Puma::TOK_NAMESPACE	`namespace`
Puma::TOK_NEQ	`!=`
Puma::TOK_NEQ_ISO_646	`not_eq`
Puma::TOK_NEW	`new`
Puma::TOK_NOT	`!`
Puma::TOK_NOT_ISO_646	`not`
Puma::TOK_OPEN_CURLY	`{`
Puma::TOK_OPEN_ROUND	`(`
Puma::TOK_OPEN_SQUARE	`[`

Token Type Constant	**Represented Text**
Puma::TOK_OPERATOR	operator
Puma::TOK_OR	|
Puma::TOK_OR_ISO_646	bitor
Puma::TOK_OR_OR	||
Puma::TOK_OR_OR_ISO_646	or
Puma::TOK_PLUS	+
Puma::TOK_POINTCUT	pointcut
Puma::TOK_PRIVATE	private
Puma::TOK_PROTECTED	protected
Puma::TOK_PTS	->
Puma::TOK_PTS_STAR	->*
Puma::TOK_PUBLIC	public
Puma::TOK_QUESTION	?
Puma::TOK_REGISTER	register
Puma::TOK_REINT_CAST	reinterpret_cast
Puma::TOK_RESTRICT	restrict
Puma::TOK_RESTRICT_2	__restrict
Puma::TOK_RESTRICT_3	__restrict__
Puma::TOK_RETURN	return
Puma::TOK_ROOF	^
Puma::TOK_ROOF_ISO_646	xor
Puma::TOK_RSH	>>
Puma::TOK_RSH_EQ	>>=
Puma::TOK_SEMI_COLON	;
Puma::TOK_SHORT	short
Puma::TOK_SIGNED	signed
Puma::TOK_SIGNED_2	__signed
Puma::TOK_SIGNED_3	__signed__
Puma::TOK_SIZEOF	sizeof
Puma::TOK_SLICE	slice
Puma::TOK_STAT_CAST	static_cast
Puma::TOK_STATIC	static
Puma::TOK_STDCALL	_stdcall
Puma::TOK_STDCALL_2	__stdcall

Token Type Constant	Represented Text
Puma::TOK_STRING_VAL	String constant like `"abc"` or `L"abc"`.
Puma::TOK_STRUCT	`struct`
Puma::TOK_SUB_EQ	`-=`
Puma::TOK_SWITCH	`switch`
Puma::TOK_TEMPLATE	`template`
Puma::TOK_THIS	`this`
Puma::TOK_THROW	`throw`
Puma::TOK_TILDE	`~`
Puma::TOK_TILDE_ISO_646	`compl`
Puma::TOK_TRY	`try`
Puma::TOK_TYPEDEF	`typedef`
Puma::TOK_TYPEID	`typeid`
Puma::TOK_TYPENAME	`typename`
Puma::TOK_TYPEOF	`typeof`
Puma::TOK_TYPEOF_2	`__typeof`
Puma::TOK_TYPEOF_3	`__typeof__`
Puma::TOK_UNION	`union`
Puma::TOK_UNKNOWN_T	`unknown_t`
Puma::TOK_UNSIGNED	`unsigned`
Puma::TOK_USING	`using`
Puma::TOK_VIRTUAL	`virtual`
Puma::TOK_VOID	`void`
Puma::TOK_VOLATILE	`volatile`
Puma::TOK_VOLATILE_2	`__volatile`
Puma::TOK_VOLATILE_3	`__volatile__`
Puma::TOK_WCHAR_T	`wchar_t`
Puma::TOK_WCHAR_T_2	`__wchar_t`
Puma::TOK_WHILE	`while`
Puma::TOK_XOR_EQ	`^=`
Puma::TOK_XOR_EQ_ISO_646	`xor_eq`
Puma::TOK_ZERO_VAL	`0`

10.5.2 Preprocessor Tokens

```
#include <Puma/PreParser.h>
```

Token Type Constant	Represented Text
TOK_PRE_IF	`#if`
TOK_PRE_ELIF	`#elif`
TOK_PRE_IFDEF	`#ifdef`
TOK_PRE_IFNDEF	`#ifndef`
TOK_PRE_ELSE	`#else`
TOK_PRE_ENDIF	`#endif`
TOK_PRE_DEFINE	`#define`
TOK_PRE_UNDEF	`#undef`
TOK_PRE_ASSERT	`#assert`
TOK_PRE_UNASSERT	`#unassert`
TOK_PRE_INCLUDE	`#include, #import`
TOK_PRE_INCLUDE_NEXT	`#include_next`
TOK_PRE_WARNING	`#warning`
TOK_PRE_ERROR	`#error`

10.5.3 White Space and Comment Tokens

```
#include <Puma/CCommentTokens.h>
```

Token Type Constant	Represented Text
Puma::TOK_WSPACE	Any white space block.
Puma::TOK_CCSINGLE	C++ style single line comment.
Puma::TOK_CCMULTIBEGIN	C style multi-line comment start token.
Puma::TOK_CCMULTIEND	C style multi-line comment end token.
Puma::TOK_CCOMMENT	Comment block.

10.6 C Grammar

trans_unit:
    decl_seq [CT_Program]
    ;

typedef_name:
    TOK_ID [CT_SimpleName]
    ;

private_name:
    ; [CT_SimpleName]

identifier:
    TOK_ID [CT_SimpleName]
    ;

literal:
    TOK_INT_VAL [CT_Integer]
    | TOK_ZERO_VAL [CT_Integer]
    | TOK_CHAR_VAL [CT_Character |CT_WideCharacter]
    | TOK_FLT_VAL [CT_Float]
    | *cmpd_str*
    ;

cmpd_str:
    str_literal
    ;

str_literal:
    TOK_STRING_VAL [CT_String |CT_WideString]
    ;

prim_expr:
    literal
    | *id_expr*
    | TOK_OPEN_ROUND *expr* TOK_CLOSE_ROUND [CT_BracedExpr]
    ;

id_expr:

TOK_ID [CT_SimpleName]

;

cmpd_literal:

TOK_OPEN_ROUND *type_id* TOK_CLOSE_ROUND

TOK_OPEN_CURLY *init_list*? TOK_CLOSE_CURLY [CT_CmpdLiteral]

;

postfix_expr:

prim_expr

| *cmpd_literal*

| (*cmpd_literal* | *prim_expr*) (

TOK_OPEN_ROUND *expr_list*? TOK_CLOSE_ROUND [CT_CallExpr]

| TOK_OPEN_SQUARE *expr* TOK_CLOSE_SQUARE [CT_IndexExpr]

| (TOK_DECR | TOK_INCR) *identifier* [CT_PostfixExpr]

| TOK_DOT *identifier* [CT_MembRefExpr]

| TOK_PTS *identifier* [CT_MembPtrExpr]

)

;

expr_list:

ass_expr

| *expr_list* TOK_COMMA *ass_expr* [CT_ExprList]

;

unary_expr:

postfix_expr

| *offsetof_expr*

| TOK_AND *cast_expr* [CT_AddrExpr]

| TOK_MUL *cast_expr* [CT_DerefExpr]

| TOK_SIZEOF (*unary_expr* | *unary_expr1*) [CT_SizeofExpr]

| (TOK_DECR | TOK_INCR) *unary_expr* [CT_UnaryExpr]

| (TOK_PLUS | TOK_MINUS | TOK_TILDE | TOK_NOT) *cast_expr* [CT_UnaryExpr]

;

unary_expr1:

TOK_OPEN_ROUND *type_id* TOK_CLOSE_ROUND

;

offsetof_expr:

TOK_OFFSETOF TOK_OPEN_ROUND *type_spec* TOK_COMMA

memb_designator TOK_OPEN_ROUND [CT_OffsetofExpr]

;

memb_designator:

identifier [CT_DesignatorSeq]

| *identifier designator+* [CT_DesignatorSeq]

;

cast_expr:

unary_expr

| (TOK_OPEN_ROUND *type_id* TOK_CLOSE_ROUND)+ *unary_expr* [CT_CastExpr]

;

mul_expr:

cast_expr

| *mul_expr* (TOK_MUL | TOK_DIV | TOK_MODULO) *cast_expr* [CT_BinaryExpr]

;

add_expr:

mul_expr

| *add_expr* (TOK_PLUS | TOK_MINUS) *mul_expr* [CT_BinaryExpr]

;

shift_expr:

add_expr

| *shift_expr* (TOK_LSH | TOK_RSH) *add_expr* [CT_BinaryExpr]

;

rel_expr:

shift_expr

| *rel_expr* (

TOK_LESS | TOK_GREATER | TOK_LEQ | TOK_GEQ

) *shift_expr* [CT_BinaryExpr]

;

equ_expr:
    rel_expr
    | *equ_expr* (`TOK_EQL` | `TOK_NEQ`) *rel_expr* [CT_BinaryExpr]
    ;

and_expr:
    equ_expr
    | *and_expr* `TOK_AND` *equ_expr* [CT_BinaryExpr]
    ;

excl_or_expr:
    and_expr
    | *excl_or_expr* `TOK_ROOF` *and_expr* [CT_BinaryExpr]
    ;

incl_or_expr:
    excl_or_expr
    | *incl_or_expr* `TOK_OR` *excl_or_expr* [CT_BinaryExpr]
    ;

log_and_expr:
    incl_or_expr
    | *log_and_expr* `TOK_AND_AND` *incl_or_expr* [CT_BinaryExpr]
    ;

log_or_expr:
    log_and_expr
    | *log_or_expr* `TOK_OR_OR` *log_and_expr* [CT_BinaryExpr]
    ;

cond_expr:
    log_or_expr
    | *log_or_expr* `TOK_QUESTION` *expr* `TOK_COLON` *cond_expr* [CT_IfThenExpr]
    ;

const_expr:
    cond_expr
    ;

ass_expr:

cond_expr
| (*unary_expr* (
TOK_ASSIGN | TOK_MUL_EQ | TOK_DIV_EQ
| TOK_MOD_EQ | TOK_ADD_EQ | TOK_SUB_EQ
| TOK_RSH_EQ | TOK_LSH_EQ | TOK_AND_EQ
| TOK_XOR_EQ | TOK_IOR_EQ
))+ *cond_expr* [CT_BinaryExpr]
;

expr:

ass_expr
| (*ass_expr* TOK_COMMA)+ *ass_expr* [CT_BinaryExpr]
;

stmt:

label_stmt
| *expr_stmt*
| *cmpd_stmt*
| *select_stmt*
| *iter_stmt*
| *jump_stmt*
;

label_stmt:

identifier TOK_COLON *stmt* [CT_LabelStmt]
| TOK_DEFAULT TOK_COLON *stmt* [CT_DefaultStmt]
| TOK_CASE *const_expr* TOK_COLON *stmt* [CT_CaseStmt]
;

expr_stmt:

expr? TOK_SEMI_COLON [CT_ExprStmt]
;

cmpd_stmt:

TOK_OPEN_CURLY *stmt_seq*? TOK_CLOSE_CURLY [CT_CmpdStmt]
;

stmt_seq:
  (*simple_decl* | *stmt*)+
  ;

select_stmt:
  TOK_SWITCH *select_stmt1 sub_stmt* [CT_SwitchStmt]
  | TOK_IF *select_stmt1 sub_stmt* [CT_IfStmt]
  | TOK_IF *select_stmt1 sub_stmt* TOK_ELSE *sub_stmt* [CT_IfElseStmt]
  ;

select_stmt1:
  TOK_OPEN_ROUND *condition* TOK_CLOSE_ROUND
  ;

sub_stmt:
  stmt
  ;

condition:
  expr
  ;

iter_stmt:
  TOK_WHILE TOK_OPEN_ROUND
  condition TOK_CLOSE_ROUND *sub_stmt* [CT_WhileStmt]
  | TOK_DO *sub_stmt* TOK_WHILE TOK_OPEN_ROUND
  expr TOK_CLOSE_ROUND TOK_SEMI_COLON [CT_DoStmt]
  | TOK_FOR TOK_OPEN_ROUND
  for_init_stmt condition? TOK_SEMI_COLON
  expr? TOK_CLOSE_ROUND *sub_stmt* [CT_ForStmt]
  ;

for_init_stmt:
  simple_decl
  | *expr_stmt*
  ;

jump_stmt:
  TOK_BREAK TOK_SEMI_COLON [CT_BreakStmt]

| `TOK_CONTINUE TOK_SEMI_COLON` [CT_ContinueStmt]
| `TOK_RETURN` *expr*? `TOK_SEMI_COLON` [CT_ReturnStmt]
| `TOK_GOTO` *identifier* `TOK_SEMI_COLON` [CT_GotoStmt]
;

decl_seq:
decl+
;

decl:
block_decl
| *fct_def*
;

block_decl:
simple_decl
| *asm_def*
;

simple_decl:
decl_spec_seq? *init_declarator_list*? `TOK_SEMI_COLON` [CT_ObjDecl]
;

decl_spec:
storage_class_spec
| *type_spec*
| *fct_spec*
| *misc_spec*
;

misc_spec:
`TOK_TYPEDEF` [CT_PrimDeclSpec]
;

decl_spec_seq:
decl_spec+ [CT_DeclSpecSeq]
;

storage_class_spec:

TOK_AUTO | TOK_REGISTER | TOK_STATIC | TOK_EXTERN [CT_PrimDeclSpec]

;

fct_spec:

TOK_INLINE [CT_PrimDeclSpec]

;

type_spec:

simple_type_spec

| *class_spec*

| *enum_spec*

| *elaborated_type_spec*

| *cv_qual*

;

simple_type_spec:

type_name

| (

TOK_CHAR | TOK_SHORT | TOK_INT

| TOK_LONG | TOK_SIGNED | TOK_UNSIGNED

| TOK_FLOAT | TOK_DOUBLE | TOK_WCHAR_T

| TOK_C_BOOL | TOK_VOID | TOK_UNKNOWN_T

) [CT_PrimDeclSpec]

;

type_name:

typedef_name

;

elaborated_type_spec:

class_key identifier [CT_ClassSpec |CT_UnionSpec]

| TOK_ENUM *identifier* [CT_EnumSpec]

;

enum_spec:

TOK_ENUM (*identifier* | *private_name*)

TOK_OPEN_CURLY *enumerator_list* TOK_CLOSE_CURLY [CT_EnumDef]

;

enumerator_list:

(*enumerator_def* TOK_COMMA)* *enumerator_def* TOK_COMMA? [CT_EnumeratorList]

;

enumerator_def:

enumerator (TOK_ASSIGN *const_expr*)? [CT_Enumerator]

;

enumerator:

identifier [CT_Enumerator]

;

asm_def:

TOK_ASM TOK_OPEN_ROUND *str_literal*

TOK_CLOSE_ROUND TOK_SEMI_COLON [CT_AsmDef]

;

init_declarator_list:

(*init_declarator* TOK_COMMA)* *init_declarator* [CT_DeclaratorList]

;

init_declarator_ext:

;

init_declarator:

declarator init_declarator_ext? *init*? [CT_InitDeclarator]

;

declarator:

direct_declarator

| *ptr_operator*+ *direct_declarator* [CT_PtrDeclarator]

;

direct_declarator:

declarator_id

| *declarator_id direct_declarator1*+ [CT_ArrayDeclarator |CT_FctDeclarator]

| TOK_OPEN_ROUND *declarator* TOK_CLOSE_ROUND [CT_BracedDeclarator]
| TOK_OPEN_ROUND *declarator*
TOK_CLOSE_ROUND *direct_declarator1*+ [CT_ArrayDeclarator |CT_FctDeclarator]
;

direct_declarator1:
TOK_OPEN_SQUARE *array_delim* TOK_CLOSE_SQUARE
| TOK_OPEN_ROUND (*identifier_list* | *param_decl_clause*) TOK_CLOSE_ROUND
;

identifier_list:
(*identifier* TOK_COMMA)* *identifier* [CT_ArgNameList]
;

array_delim:
cv_qual_seq? (TOK_MUL | *ass_expr*)? [CT_ArrayDelimiter]
| TOK_STATIC *cv_qual_seq*? *ass_expr* [CT_ArrayDelimiter]
| *cv_qual_seq* TOK_STATIC *ass_expr* [CT_ArrayDelimiter]
;

ptr_operator:
TOK_MUL *cv_qual_seq*?
;

cv_qual_seq:
cv_qual+ [CT_DeclSpecSeq]
;

cv_qual:
TOK_CONST | TOK_VOLATILE | TOK_RESTRICT [CT_PrimDeclSpec]
;

declarator_id:
identifier
;

type_id:
type_spec_seq (*abst_declarator* | *private_name*) [CT_NamedType]
;

type_spec_seq:

type_spec+ [CT_DeclSpecSeq]

;

abst_declarator:

direct_abst_declarator

| *ptr_operator*+ *direct_abst_declarator*? [CT_PtrDeclarator]

;

direct_abst_declarator:

direct_abst_declarator1

| *direct_abst_declarator1 direct_abst_declarator1*+ [CT_ArrayDeclarator |CT_FctDeclarator]

| `TOK_OPEN_ROUND` *abst_declarator* `TOK_CLOSE_ROUND` [CT_BracedDeclarator]

| `TOK_OPEN_ROUND` *abst_declarator* `TOK_CLOSE_ROUND` *direct_abst_declarator1*+ [CT_ArrayDeclarator |CT_FctDeclarator]

;

direct_abst_declarator1:

`TOK_OPEN_ROUND` *param_decl_clause*? `TOK_CLOSE_ROUND`

| `TOK_OPEN_SQUARE` (*ass_expr* | `TOK_MUL`)? `TOK_CLOSE_SQUARE`

;

param_decl_clause:

(*param_decl_list* `TOK_ELLIPSIS`?)? [CT_ArgDeclList]

;

param_decl_list:

(*param_decl* `TOK_COMMA`)* *param_decl* `TOK_COMMA`?

;

param_decl:

decl_spec_seq (*abst_declarator* | *declarator* | *private_name*) [CT_ArgDecl]

;

fct_def:

decl_spec_seq? *declarator arg_decl_seq*? *fct_body* [CT_FctDef]

;

arg_decl_seq:

    simple_decl+ [CT_ArgDeclSeq]

    ;

fct_body:

    cmpd_stmt

    ;

init:

    `TOK_ASSIGN` *init_clause* [CT_ExprList]

    ;

init_clause:

    ass_expr

    | `TOK_OPEN_CURLY` *init_list* `TOK_CLOSE_CURLY` [CT_ExprList]

    ;

init_list:

    (*init_list_item* `TOK_COMMA`)* *init_list_item* `TOK_COMMA`? [CT_ExprList]

    ;

init_list_item:

    init_clause

    | *designation init_clause* [CT_BinaryExpr]

    ;

designation:

    designator+ `TOK_ASSIGN` [CT_DesignatorSeq]

    ;

designator:

    `TOK_DOT` *identifier* [CT_MembDesignator]

    | `TOK_OPEN_SQUARE` *const_expr* `TOK_CLOSE_SQUARE` [CT_IndexDesignator]

    ;

class_spec:

    class_head `TOK_OPEN_CURLY` *member_spec*?

    `TOK_CLOSE_CURLY` [CT_UnionDef |CT_ClassDef]

    ;

class_head:

class_key (*identifier* | *private_name*)

;

class_key:

TOK_STRUCT | TOK_UNION [CT_Token]

;

member_spec:

member_decl+ [CT_MembList]

;

member_decl:

type_spec_seq member_declarator_list TOK_SEMI_COLON [CT_ObjDecl]

;

member_declarator_list:

(*member_declarator* TOK_COMMA)* *member_declarator* [CT_DeclaratorList]

;

member_declarator:

declarator [CT_InitDeclarator]

| (*declarator* | *private_name*) TOK_COLON *const_expr* [CT_BitFieldDeclarator]

;

10.7 C++ Grammar

10.8 Preprocessor Grammar

Index