FramaCIRGen  
a clang → frama-c/cabs converter

Writing a parser and a type-checker for languages as complex as C++ is not an easy task. For the tool “FramaCIRGen” it was thus necessary to build upon existing front-ends and transform their output so that it can be accepted by Frama-C. We have chosen clang[[1]](#footnote-1) as the reference STANCE C++ front-end, because of its robustness, its licence, its easy customization and its current and active evolutions.

Clang is the C/C++/Objective-C/Swift front-end of the LLVM[[2]](#footnote-2) compiler. It is available under an Open-Source license[[3]](#footnote-3),[[4]](#footnote-4) compatible with the industrial purposes of the tools. The LLVM and clang designs are very modular, and a high-level intermediate representation is available and relatively well documented. In addition, clang is widely used and enjoys a very active development community, so that support and maintenance should be available on the longer run.

For the complete translation chain, we have retained the following design (see the Figure 1 below).

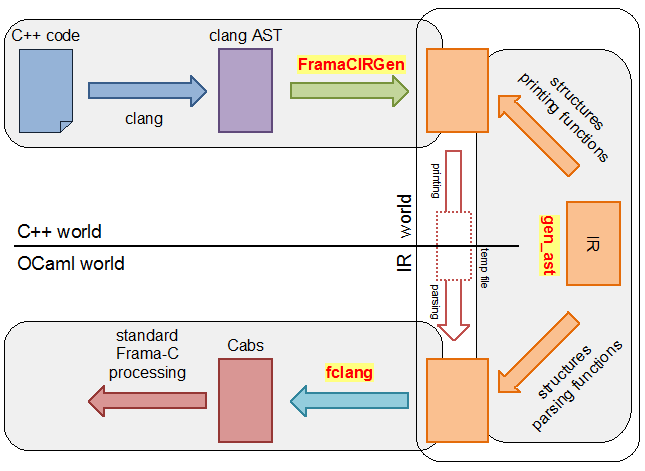


Figure 1: design of the front-end integration

Apart from language support, some features are required from the front-ends in order to ease their usage. First, the locations in the original source file of each construction must be collected with as much precision as possible (note that the usage of macros makes it difficult to attain the column number level), and preserved across the translations from one AST to the other. This is indeed key in enabling traceability between the normalized AST on which the analyzers operate and the original code. Second, macros must be expanded in formal specifications as well as in the code. Namely, standard pre-processors do not perform macro expansion in comments, which is what specifications are in the first place in order not to interfere with normal compilation process. Some mechanism similar to what exists in Frama-C for C macros is thus required. From a software-engineering point of view, it is indeed important to be able to use macros, for instance for architecture- or configuration- dependent constants, in specifications. Last, the C++ front-end must be designed so as to be easily plugged on top of Frama-C.

The central part of the design is the intermediate representation (IR world of Figure 1). As clang is written in C++ and Frama-C is written in OCaml, we have chosen to implement two units to make the connection between these two worlds: “FramaCIRGen” is an executable based on clang that generates a temp file in the IR format from the source code. The “fclang” Frama-C plugin reads the temp file and produces all the necessary information in the internal Frama-C/cabs format for the needs of the various other plugins.

To ensure the consistency between the writing of C++ functions in the IR format of “FramaCIRGen” and the reading of OCaml functions from the IR Format of the “fclang” Frama-C plugin, we have written the structure of C++ constructions in a dedicated language. Then, a generator “gen\_ast” generates the files

* intermediate\_format.h, intermediate\_format.c to be used by “FramaCIRGen”
* intermediate\_format.mli, intermediate\_format\_parser.mli, intermediate\_format\_parser.ml to be used by the “fclang” Frama-C plugin

Our documentation exclusively describes the construction of “FramaCIRGen”, the executable based on clang that generates a temp file in the IR format from the source code. The figure below defines the part of **Erreur ! Source du renvoi introuvable.**

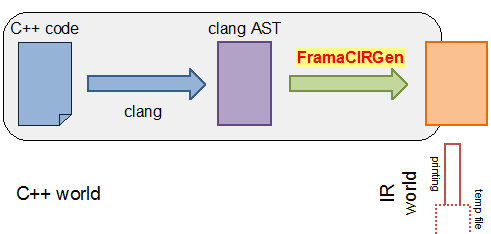


Figure 2: FramaCIRGen part of Figure 1

This documentation is organized around the C++ classes and their methods. It has adopted a bottom-up organization from the basic low-level concepts to the higher-level concepts.

The main input of “FramaCIRGen” is the clang AST and its output is the calls to the API (Application Programming Interface) defined by the construction functions of the generated file “intermediate\_format.h”. These calls build an intermediate tree that is output at the end in the temp file by the function output\_file declared in “intermediate\_format.h”.

“FramaCIRGen” works in two steps:

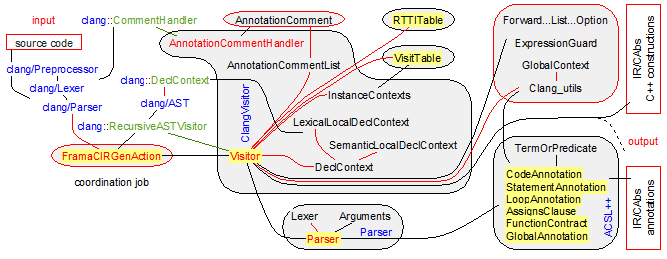
* The first step occurs during the clang parsing of the C++ source code. clang enables to register comment handlers for specific treatments. We have implemented specific handlers to register the ACSL++ comments towards their associated C++ constructions. For example, function contracts are recognized thanks to the first keywords of the comments – like requires, ensures, assumes, behavior, … Preprocessing of macros within the ACSL++ comments should be done during this step. For the moment it is not the case and macros are mainly handled in the second step.
* The second and main step consists in visiting the complete clang AST. During this step, we call the construction functions of the generated file “intermediate\_format.h”. At the end of this step, we simply call the function output\_file to write the built intermediate tree in the temp file. The annotations are actually parsed during the visit.

This documentation is organized around 3 main units. The first unit FramaCIRGenAction (3.2kloc) provides the material for the first step. The second unit ClangVisitor (11kloc) describes the translation of the C++ source code and the last third unit ACSL++ (21kloc) describes the translation of the annotations.

The translation of the C++ source code into the intermediate format (defined in the file “intermediate\_format.ast” and close to the Frama-C/Cabs format) has required two key technologies. The first technology is the translation of the object mechanism (inheritance, virtual methods) into a Cabs interpretable format. It is defined in the RTTITable unit. All other translations are rather genuine and direct. The second technology consists in ensuring that all the generated constructions are at least declared before being used. All would have been perfect if C++ had no template mechanism. In fact, clang visits the implicit instance classes and methods in the scope of their associated template classes and methods. Hence an instance class C<Integer> could be visited before the class Integer and the dependency relations could be even more intricate. To generate the declarations before they are used, the unit VisitTable delays the generation of some instance clang AST node to the visit of all the clang AST nodes it depends.

The translation of the ACSL++ annotations is the biggest part of FramaCIRGen. It is based on an embedded and customized recursive descent lexer/parser defined in the Parser unit. Then the unit ACSL++ defines the ACSL++ grammar that produces the intermediate format corresponding to the annotations in “intermediate\_format.ast”. The key point of this grammar is the parsing of the terms, the predicates, the sets and the ranges with left/reduce technology because the descending context does not permit to know the kind of construct that is parsed before it is actually parsed.

The global graph of classes is summarized on the schema below:



The FramaCIRGenAction Unit

The FramaCIRGenAction unit performs the coordination job for the other main units ClangVisitor and ACSL++. Its main function is to prepare the visit of the clang AST within the clang world.

To visit the clang AST, the key points consists in:

1. defining a clang::ASTFrontendAction that overrides the method clang::ASTFrontendAction::CreateASTConsumer. This clang::ASTFrontendAction is active during the preprocessing, the parsing and the visit of the AST.
2. defining a comment handler clang::CommentHandler to register the ACSL++ comments towards their associated C++ constructions. For example, function contracts are recognized thanks to the first keywords of the comments – like requires, ensures, assumes, behavior, … Preprocessing of macros within the ACSL++ comments should be done during this step. For the moment it is not the case and macros are mainly handled during the visit of the clang AST.
3. defining a clang AST visitor that inherits from clang::ASTConsumer and from clang::RecursiveASTVisitor. The overridden method of clang::ASTFrontendAction::CreateASTConsumer returns this visitor. Such a visitor is created before the preprocessing step. The preprocessing and the parsing are likely to fill some fields of the visitor but it is not really active. Then the virtual method of the visitor that implements clang::ASTConsumer::HandleTranslationUnit is called. The main job of this method consists in calling clang::RecursiveASTVisitor::TraverseDecl with the root node of the clang AST built by the clang parser. This method visits all the nodes of the AST and its objective is to call the adequate construction function of “intermediate\_format.h” to build a new AST in the IR. For that, when the visit encounters adequate clang nodes, it automatically calls the appropriate abstract methods among clang::RecursiveASTVisitor::VisitNamespaceDecl, clang::RecursiveASTVisitor::VisitRecordDecl, clang::RecursiveASTVisitor::VisitClassTemplatePartialSpecializationDecl, clang::RecursiveASTVisitor::VisitEnumDecl, clang::RecursiveASTVisitor::VisitTypedefNameDecl, clang::RecursiveASTVisitor::VisitFieldDecl, clang::RecursiveASTVisitor::VisitVarDecl, clang::RecursiveASTVisitor::VisitFunctionDecl. Hence these abstract methods should be implemented in our visitor.

The justification of this implementation comes from the clang documentation of clang::RecursiveASTVisitor:

Clients of this visitor should subclass the visitor (providing themselves as the template argument, using the curiously recurring template pattern) and override any of the clang::RecursiveASTVisitor::Traverse\*, clang::RecursiveASTVisitor::WalkUpFrom\*, and Visit\* methods for declarations, types, statements, expressions, or other AST nodes where the visitor should customize behavior. Most users only need to override Visit\*. Advanced users may override clang::RecursiveASTVisitor::Traverse\* and clang::RecursiveASTVisitor::WalkUpFrom\* to implement custom traversal strategies. Returning false from one of these overridden functions will abort the entire traversal.

By default, this visitor tries to visit every part of the explicit source code exactly once. The default policy towards templates is to descend into the 'pattern' class or function body, not any explicit or implicit instantiations. Explicit specializations are still visited, and the patterns of partial specializations are visited separately. This behavior can be changed by overriding clang::RecursiveASTVisitor::shouldVisitTemplateInstantiations() in the derived class to return true, in which case all known implicit and explicit instantiations will be visited at the same time as the pattern from which they were produced.

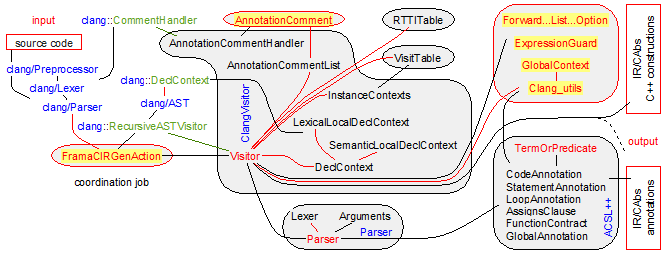
The current implementation of FramaCIRGen follows these guidelines. The entry class is the class FramaCIRGenAction that inherits from clang::ASTFrontendAction to provide an implementation for clang::ASTFrontendAction::CreateASTConsumer (see guideline 1). This implementation creates a ClangVisitor::Visitor that inherits from clang::ASTConsumer and from clang::RecursiveASTVisitor. The ClangVisitor::Visitor stores a ClangVisitor::AnnotationCommentHandler that inherits from clang::CommentHandler. The ClangVisitor::AnnotationCommentHandler is registered at the level of the clang::Preprocessor and is active during the preprocessing phase (see guideline 2). When the preprocessor encounters a comment that it recognizes as an ACSL++ comment it stores it into ClangVisitor::AnnotationCommentList. At the end of the preprocessing and of the parsing phase, the complete clang/AST is defined in a clang::Scope, the ACSL++ comments are attached to their right nodes in the clang/AST. Then the next step starts when FramaCIRGenAction (indirectly) calls the methods ClangVisitor::Visitor::HandleTranslationUnit.

ClangVisitor::Visitor::HandleTranslationUnit executes the complete visit of all the clang nodes stored in clang::Scope (see guideline 3). During this visit, it manages

* the lexical context of clang with the class ClangVisitor::LexicalLocalDeclContext (see the hierarchic link clang::Decl::getLexicalParent()) and the semantic context of clang with the class ClangVisitor::SemanticLocalDeclContext (see the hierarchic link clang::Decl::getParent()),
* the creation of the Virtual Method Tables with the RTTITable::RTTITable class,
* the generation in the right order of the instances of the template classes with the classes VisitTable::VisitTable and ClangVisitor::InstanceContexts,
* the generation of the C++ constructions in the Intermediate Representation (see the file intermediate\_format.ast) in connection with the Frama-C/Cabs format thanks to local helper classes (Clang\_utils, Forward…List…Option, ExpressionGuard) for this format,
* the generation of the ACSL++ annotations in the Intermediate Representation in connection with the Frama-C/Cabs format thanks to a dedicated Parser::Parser engine. Depending from the clang/AST node they depend from, the parser manages the generation of the annotations with the ACSL++ main rules defined by the classes ACSL++::CodeAnnotation, ACSL++::StatementAnnotation, ACSL++::LoopAnnotation, ACSL++::AssignsAnnotation, ACSL++::FunctionContract, ACSL++::GlobalAnnotation. All the annotations contain terms and predicates and they should recognize the identifiers in them as identifiers in the C++ world of identifiers in the logic world. This job is done by the Clang\_utils and the GlobalContext classes – GlobalContext stores the global logic definitions defined as logic types, logic functions, logic predicates, lemmas and axioms.

At the end of the visit, ClangVisitor::Visitor::HandleTranslationUnit has built a complete Intermediate Representation (see the file intermediate\_format.ast). It just has to persists it in the output temp file thanks to the output\_file function.

The graph classes of the FramaCIRGenAction unit is defined on the schema below:



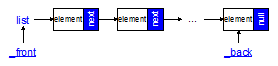
The classes ForwardReferenceList, ForwardList and ForwardListOption

The intermediate format defined in the file “intermediate\_format.ast” is a hierarchic format based on a tree representation. It is dedicated to persistence: trees are written and read very quickly in a linear way without any need of additional table. Hence the data structures defined in “intermediate\_format.ast” are struct, flagged union, strings (**char**\*), single linked lists (**typedef** list) and option (**typedef** option). Hence single linked lists have no backward link and we need additional information to insert at the end of the list, which is the natural insertion position.

The classes ForwardReferenceList, ForwardList and ForwardListOption are class helpers to manage single linked lists (**typedef** list) in various contexts. The class ForwardReferenceList manages the single linked list with an additional pointer at the end of the list. This additional pointer is the natural insertion position in the list. The class ForwardList performs the same job, but in the case of ForwardReferenceList the **typedef** list is outside the class whereas in the case of ForwardList the **typedef** list is inside the class. The class ForwardListOption manages an optional list that practically represents the declarations inside a namespace or inside a class.

The class ForwardReferenceList is the most useful class. It is used in every class in contact with the intermediate representation defined by “intermediate\_format.ast”. This concerns the class ClangVisitor::Visitor, all classes in the unit ACSL++, ClangVisitor::DeclContext, RTTITable::RTTITable, VisitTable::VisitTable. The class ForwardList is mainly used in ACSL++ for ACSL++::FunctionContract, ACSL++::GlobalAnnotation, ACSL++::AssignsClause, ACSL++::LoopAnnotation, ACSL++::StatementAnnotation and in ClangVisitor::Visitor for the generation of delayed implementations that are waiting for a translation unit context. The class ForwardListOption is only used in ClangVisitor::LexicalLocalDeclContext.

A ForwardReferenceList is represented on the schema:



and is used as following

*/\* statement \*/* list body = **NULL**;

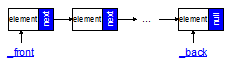
ForwardReferenceList lbody(body);

lbody.insertContainer(statement\_Expression(...));

lbody.insertContainer(statement\_While(...));

*/\* body now contains two elements, first an expression statement and second a while statement \*/*

A ForwardList is represented on the schema:



and is used as following

ForwardList lbody;

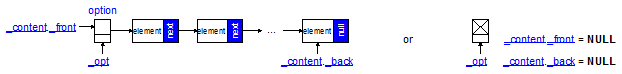
lbody.insertContainer(statement\_Expression(...));

lbody.insertContainer(statement\_While(...));

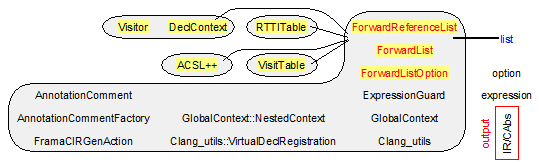
*/\* statement \*/* list body = lbody.getFront();

*/\* body now contains two elements, first an expression statement and second a while statement \*/*

A ForwardListOption is represented on the schema:



The class environment of these classes is defined on the schema below:



Fields of the class ForwardReferenceList

list\* \_front;

Reference to the list in which we insert some elements. \_front ≠ **NULL**.

list \_back;

last cell of the list. This field is **NULL** if and only if the \_front list is empty.

Invariants

The following invariants should hold:

* \_front ≠ **NULL**,
* the list is empty and \*\_front = \_back = **NULL**,
* the list is non-empty and \*\_front ≠ **NULL**, back ≠ **NULL** and \_back->next = **NULL**,
* the list contains only one element and \*\_front = \_back ≠ **NULL**,
* more generally the list contains n > 0 elements and \_back = (\*\_front)(->next)n-1.

Fields of the class ForwardList

list \_front;

list in which we insert some elements. We should have the ownership on this list.

list \_back;

last cell of the list \_front. This field is **NULL** if and only if the \_front list is empty.

Invariants

The following invariants should hold:

* the list is empty and \_front = \_back = **NULL**,
* the list is non-empty and \_front ≠ **NULL**, back ≠ **NULL** and \_back->next = **NULL**,
* the list contains only one element and \_front = \_back ≠ **NULL**,
* more generally the list contains n > 0 elements and \_back = \_front(->next)n-1.

Fields of the class ForwardListOption

option\* \_opt;

Reference to the option list in which we are likely to insert elements. \_opt ≠ **NULL**.

ForwardReferenceList \_content;

Short-circuit to the list stored in \_opt->content.container when this place is valid.

Declaration of the classes ForwardReferenceList, ForwardList and ForwardListOption

**class** ForwardReferenceList {

**private**:

list\* \_front;

list \_back;

**public**:

ForwardReferenceList() : \_front(**NULL**), \_back(**NULL**) {}

ForwardReferenceList(**const** ForwardReferenceList& source) = **default**;

ForwardReferenceList(list& alist);

**bool** isValid() **const** { **return** \_front && (!\*\_front ? !\_back : (\_back && !\_back->next)); }

**void** clear();

ForwardReferenceList& insertPlain(**long** value)

ForwardReferenceList& insertContainer(**void**\* value);

ForwardReferenceList& append(ForwardReferenceList& tail);

ForwardReferenceList& append(ForwardList& tail);

list getBack() **const** { **return** \_back; }

list& getFront() **const** { **assert**(\_front); **return** \*\_front; }

};

**class** ForwardList {

**private**:

list \_front;

list \_back;

**public**:

ForwardList() : \_front(**NULL**), \_back(**NULL**) {}

ForwardList(**const** ForwardList& source) = **default**;

**bool** isValid() **const** { **return** (!\_front ? !\_back : (\_back && !\_back->next)); }

ForwardList& insertPlain(**long** value);

ForwardList& insertContainer(**void**\* value);

ForwardList& append(**const** ForwardList& tail);

ForwardList& append(list tail);

list getFront() **const** { **return** \_front; }

};

**class** ForwardListOption {

**private**:

option\* \_opt;

ForwardReferenceList \_content;

**public**:

ForwardListOption() : \_opt(**NULL**), \_content() {}

ForwardListOption(list& content) : \_opt(**NULL**), \_content(content) {}

ForwardListOption(option& aopt);

**bool** isValid() **const** { **return** \_opt ? ((\*\_opt)->is\_some?\_content.isValid():**true**) : \_content.isValid(); }

list getCore() **const** { **assert**(\_content.isValid()); **return** \_content.getBack(); }

ForwardListOption& insertPlain(**long** value);

ForwardListOption& insertContainer(**void**\* value);

ForwardReferenceList& getList();

};

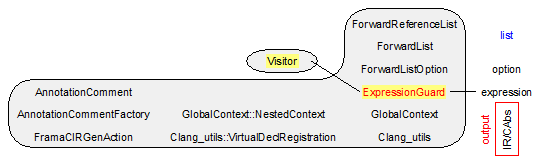
The class ExpressionGuard

The class ExpressionGuard is a class helper that automatically manages the initialization of variables and the construction of objects. For that it keeps in its field \_thisStarExpr the expression that should be assigned.

The standard way to use it is to create such a guard that keeps \_thisStarExpr during the visit of its initialization. Then the definition expr of type clang::Expr is visited and it produces an exp\_node. At the end the method setAssignResult is call to perform the assignment in the Intermediate Representation defined by the file “intermediate\_format.ast”.

This class is exclusively used in ClangVisitor::Visitor::makeExpression and in the methods it calls. Sometimes, an expression is visited, but it has no variable to be assigned in. In such a case \_thisStarExpr remains **NULL** and nothing happens. Sometimes, the assignment requires a more complex management and the method releaseExpr is called to perform this treatment. The destructor guarantees that our class has the ownership onto \_thisStarExpr at least until the actual assignment occurs.

The class environment of ExpressionGuard is defined on the schema below:



Declaration of the class ExpressionGuard

**class** ExpressionGuard {

**private**:

**const** ClangVisitor::Visitor& \_visitor;

expression \_thisStarExpr;

**public**:

ExpressionGuard(**const** ClangVisitor::Visitor& visitor, expression& thisStarExpr)

: \_visitor(visitor), \_thisStarExpr(thisStarExpr) { thisStarExpr = **NULL**; }

~ExpressionGuard() { **if** (\_thisStarExpr) free\_expression(\_thisStarExpr); }

exp\_node setAssignResult(exp\_node result, **const** clang::Expr\* expr)

{ **if** (\_thisStarExpr)

{ location loc = \_visitor.makeLocation(expr->getSourceRange());

result = exp\_node\_Assign(\_thisStarExpr, expression\_cons(loc,result));

\_thisStarExpr = **NULL**;

};

**return** result;

}

expression releaseExpr() { expression result = \_thisStarExpr; **assert**(result); \_thisStarExpr = **NULL**; **return** result; }

};

The class GlobalContext::NestedContext

The ACSL++ annotations usually contain terms and predicates and they should recognize the identifiers in them as identifiers in the C++ world of identifiers in the logic world. This job is done thanks to the GlobalContext class that acts as a hierarchic dictionary. GlobalContext stores the global logic definitions defined as logic types, logic functions, logic predicates, lemmas and axioms.

The class NestedContext represents the words and their definition in this dictionary. The definitions often refer to the output intermediate format defined by the file “intermediate\_format.ast”. The more natural way to integrate the definition when an identifier is recognized as a NestedContext is to duplicate this definition thanks to the dup\_… functions.

The hierarchy is managed with specific NestedContext that have sons. This hierarchy corresponds to the namespaces (class Qualification), the classes (class Qualification), and the instances of template classes (class TemplateQualification) in which the global definitions have been defined.

Note that a TemplateQualification is always defined as a soon of the class Qualification whose Qualification::tag = QTEMPLATEINSTANCE. The TemplateQualification part defines the template instance arguments and the Qualification part defines the name. This is consistent with the fact that a template class may be instantiated multiple times with different template instance arguments.

When the user declares a new logic type like

**type** list\_integer = Nil | Cons (integer, list\_integer);

with the following ACSL++ type construction:

logic-def ::= **type** logic-type = logic-type-def ;

logic-type-def ::= basic-type | user-defined-type | record-type | sum-type | product-type ;

this creates a LogicType in the GlobalContext dictionary associated with its definition.

This also creates two LogicConstructor in the dictionary: the first one is associated to Nil and the second one to Cons.

When the user declares a new logic variable with the following ACSL++ type construction:

logic-const-def ::= **logic** type-expr poly-id = term ;

this creates a LogicVariable in the GlobalContext dictionary associated with its name and its kind.

When the user declares a new logic function/predicate like

**predicate** is\_positive (integer x) = x > 0;

**logic** integer get\_sign (real x) = x > 0.0 ? 1 : ( x < 0.0 ? -1 : 0);

with the following ACSL++ type construction:

logic-function-def ::= **logic** type-expr poly-id parameters = term ;

predicate-def ::= **predicate** poly-id parameters? = pred ;

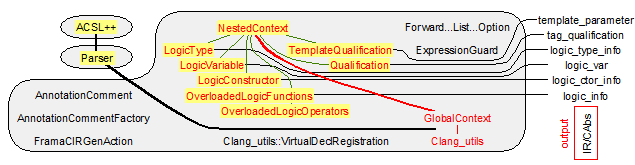
this creates a OverloadedLogicFunctions in the GlobalContext dictionary associated with its name and its signature. If a function/predicate of the same name but with a different signature soon exists, the previous OverloadedLogicFunctions is preserved and the new function/predicate is added alongside. The overloading is an ACSL++ extension of ACSL. ACSL++ also supports operators as an ACSL++ extension of ACSL. An operator is simply a function declared with the keyword **operator**.

**type** complex = (real, real) ;

**logic** complex **operator**+ (complex x, complex y) = (fst(x) + fst(y), snd(x) + snd(y));

This creates an OverloadedLogicOperators (that inherits from OverloadedLogicFunctions) in the GlobalContext dictionary associated with its name and its signature.

The class environment of NestedContext is defined on the schema below:



Declaration of the class GlobalContext::NestedContext

**class** NestedContext {

**private**:

NestedContext\* \_parent;

std::string \_name;

**public**:

NestedContext(**const** std::string& name) : \_parent(**NULL**), \_name(name) {}

NestedContext(**const** NestedContext& source) : \_parent(**NULL**), \_name(source.\_name) {}

**virtual** ~NestedContext() {}

**const** std::string& getName() **const** { **return** \_name; }

**enum** Type { TUndefined, TLogicVariable, TOverloadedLogicFunctions, TLogicType, TLogicConstructor, TQualification, TTemplateQualification };

**virtual** Type getType() **const** { **return** TUndefined; }

**bool** isLogicVariable() **const** { **return** getType() == TLogicVariable; }

**bool** isOverloadedLogicFunctions() **const** { **return** getType() == TOverloadedLogicFunctions; }

**bool** isLogicType() **const** { **return** getType() == TLogicType; }

**bool** isLogicConstructor() **const** { **return** getType() == TLogicConstructor; }

**bool** isQualification() **const** { **return** getType() == TQualification; }

**bool** isTemplateQualification() **const** { **return** getType() == TTemplateQualification; }

LogicVariable& asLogicVariable() { **assert**(getType() == TLogicVariable); **return** (LogicVariable&) \***this**; }

**const** LogicVariable& asLogicVariable() **const** { **assert**(getType() == TLogicVariable); **return** (**const** LogicVariable&) \***this**; }

OverloadedLogicFunctions& asOverloadedLogicFunctions()

{ **assert**(getType() == TOverloadedLogicFunctions); **return** (OverloadedLogicFunctions&) \***this**; }

**const** OverloadedLogicFunctions& asOverloadedLogicFunctions() **const**

{ **assert**(getType() == TOverloadedLogicFunctions); **return** (const OverloadedLogicFunctions&) \***this**; }

LogicType& asLogicType() { **assert**(getType() == TLogicType); **return** (LogicType&) \***this**; }

**const** LogicType& asLogicType() **const** { **assert**(getType() == TLogicType); **return** (**const** LogicType&) \***this**; }

LogicConstructor& asLogicConstructor() { **assert**(getType() == TLogicConstructor); **return** (LogicConstructor&) \***this**; }

**const** LogicConstructor& asLogicConstructor() **const** { **assert**(getType() == TLogicConstructor); **return** (**const** LogicConstructor&) \***this**; }

Qualification& asQualification() { **assert**(getType() == TQualification); **return** (Qualification&) \***this**; }

**const** Qualification& asQualification() **const** { **assert**(getType() == TQualification); **return** (const Qualification&) \***this**; }

TemplateQualification& asTemplateQualification() { **assert**(getType() == TTemplateQualification); **return** (TemplateQualification&) \***this**; }

**const** TemplateQualification& asTemplateQualification() **const**

{ **assert**(getType() == TTemplateQualification); **return** (**const** TemplateQualification&) \***this**; }

NestedContext\* sparent() **const** { **return** \_parent; }

**void** setParent(NestedContext\* parent) { **assert**(!\_parent); \_parent = parent; }

**bool** hasParent() **const** { **return** \_parent != **NULL**; }

**bool** hasSoons() **const** { **return** **const\_cast**<NestedContext\*>(**this**)->ssons() != **NULL**; }

**struct** Less {

**bool** **operator**()(**const** NestedContext\* c1, **const** NestedContext\* c2) **const**

{ **if**(!c1) **return** c2; *// NULL is less than everything*

**if**(!c2) **return** **false**;

**return** c1->compare(\*c2) < 0;

}

};

**virtual** **int** compare(**const** NestedContext& c) **const** { **return** **this**->\_name.compare(c.\_name); }

**typedef** std::set<NestedContext\*, Less> SoonsSet;

**virtual** SoonsSet\* ssons() { **return** **NULL**; }

SoonsSet& sons() { SoonsSet\* result = ssons(); **assert**(result); **return** \*result; }

**const** SoonsSet& sons() **const** { SoonsSet\* result = **const\_cast**<NestedContext\*>(**this**)->ssons(); **assert**(result); **return** \*result; }

};

**class** LogicVariable : **public** NestedContext {

**private**:

**typedef** NestedContext inherited;

logic\_var lvVariable;

**public**:

LogicVariable(**const** std::string& name, logic\_var variable) : inherited(name), lvVariable(variable) {}

**virtual** ~LogicVariable() { free\_logic\_var(lvVariable); }

**virtual** Type getType() **const** { **return** TLogicVariable; }

};

**class** OverloadedLogicFunctions : **public** NestedContext {

**public**:

**typedef** std::list<std::pair<**bool**, logic\_info> > Functions;

**private**:

**typedef** NestedContext inherited;

Functions \_logicFunctions; *// the first bool is the method flag*

**public**:

OverloadedLogicFunctions(**const** std::string& name, logic\_info info, **bool** isMethod=**false**)

: inherited(name) { \_logicFunctions.push\_back(std::make\_pair(isMethod, info)); }

**virtual** ~OverloadedLogicFunctions();

**virtual** Type getType() **const** { **return** TOverloadedLogicFunctions; }

**virtual** **bool** isOperator() **const** { **return** **false**; }

OverloadedLogicOperators& asOperator();

**void** addFunction(logic\_info info, **bool** isMethod=**false**) { \_logicFunctions.push\_back(std::make\_pair(isMethod, info)); }

**const** Functions& getFunctions() **const** { **return** \_logicFunctions; }

};

**class** OverloadedLogicOperators : **public** OverloadedLogicFunctions {

**private**:

**typedef** OverloadedLogicFunctions inherited;

**int** \_codeOperator;

**public**:

OverloadedLogicOperators(**const** std::string& name, **int** codeOperator, logic\_info info, **bool** isMethod=**false**)

: inherited(name, info, isMethod), \_codeOperator(codeOperator) {}

**virtual** **bool** isOperator() **const** { **return** **true**; }

**int** getCodeOperator() **const** { **return** \_codeOperator; }

};

**class** LogicType : **public** NestedContext {

**private**:

**typedef** NestedContext inherited;

logic\_type\_info \_type;

**public**:

LogicType(**const** std::string& name, logic\_type\_info type) : inherited(name), \_type(type) {}

**virtual** ~LogicType() { free\_logic\_type\_info(\_type); }

**virtual** Type getType() **const** { **return** TLogicType; }

logic\_type\_info type\_info() **const** { **return** \_type; }

};

**class** LogicConstructor : **public** NestedContext {

**private**:

**typedef** NestedContext inherited;

logic\_ctor\_info lciConstructor;

**public**:

LogicConstructor(**const** std::string& name, logic\_ctor\_info constructor) : inherited(name), lciConstructor(constructor) {}

**virtual** ~LogicConstructor() { free\_logic\_ctor\_info(lciConstructor); }

**virtual** Type getType() **const** { **return** TLogicConstructor; }

logic\_ctor\_info getInfo() **const** { **return** lciConstructor; }

};

**class** Qualification : **public** NestedContext {

**private**:

**typedef** NestedContext inherited;

NestedContext::SoonsSet mSoons;

tag\_qualification tag;

**public**:

Qualification(**const** std::string& name, tag\_qualification t) : inherited(name), tag(t) {}

**virtual** ~Qualification();

**virtual** Type getType() **const** { **return** TQualification; }

**virtual** NestedContext::SoonsSet\* ssons() { **return** &mSoons; }

**bool** hasRecordType() **const** { **return** tag == QSTRUCTORCLASS; }

**bool** hasTemplateRecordType() **const** { **return** tag == QTEMPLATEINSTANCE; }

TemplateQualification\* findInstance(*/\* template*\_parameter *\*/* list parameters) **const**;

qualified\_name makeRecordName() **const**;

*/\* qualification \*/* list makeQualificationList() **const**;

qualification getQualification() **const**;

};

**class** TemplateQualification : **public** NestedContext {

**private**:

**typedef** NestedContext inherited;

NestedContext::SoonsSet mSoons;

*/\* template*\_parameter *\*/* list \_parameters;

**public**:

TemplateQualification(*/\* template*\_parameter *\*/* list parameters);

**virtual** ~TemplateQualification();

*/\* qualification \*/* list makeQualificationList() **const**;

qualification getQualification(**const** **char**\* name) **const**;

**virtual** **int** compare(**const** NestedContext& c) **const**;

**virtual** Type getType() **const** { **return** TTemplateQualification; }

**virtual** NestedContext::SoonsSet\* ssons() { **return** &mSoons; }

*/\* template\_parameter \*/* list getParameters() **const** { **return** \_parameters; }

};

The class GlobalContext

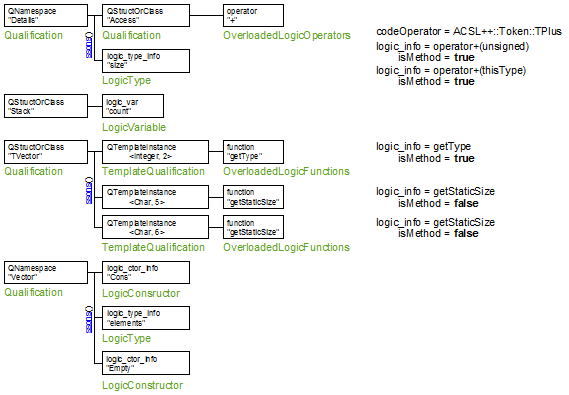
The ACSL++ annotations usually contain terms and predicates and they should recognize the identifiers in them as identifiers in the C++ world of identifiers in the logic world. This job is done thanks to the GlobalContext class that acts as a hierarchic dictionary. GlobalContext stores the global logic definitions defined as logic types, logic functions, logic predicates, lemmas and axioms.

The class NestedContext represents the words and their definition in this dictionary. The definitions often refer to the output intermediate format defined by the file “intermediate\_format.ast”. The more natural way to integrate the definition when an identifier is recognized as a NestedContext is to duplicate this definition thanks to the dup\_… functions.

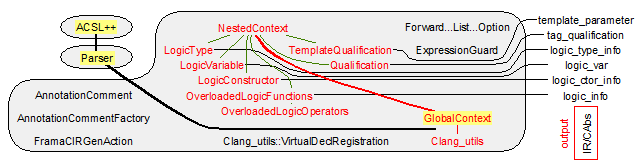
The hierarchy is managed with specific NestedContext that have sons. This hierarchy corresponds to the namespaces (class Qualification), the classes (class Qualification), and the instances of template classes (class TemplateQualification) in which the global definitions have been defined.

The documentation of NestedContext gives more examples about the content of our GlobalContext. Our dictionary is a field of Clang\_utils and this more global class is able to give the exact meaning of any identifier used in ACSL (identifier in the logic world if it is in our dictionary or identifier in the C++ world if it is in the current C++ scope).

An object of the class GlobalContext has the following representation:



The class environment of GlobalContext is defined on the schema below:



Fields of the class GlobalContext

**int** \_variableNumber;

Counter to enable the creation of fresh logic variable.

NestedContext::SoonsSet \_logicTable;

Forest that defines the hierarchic dictionary of NestedContext. At the first level \_logicTable contains Qualification nodes and many leaves of type LogicType, LogicVariable, LogicConstructor, OverloadedLogicFunctions (that are not methods) OverloadedLogicOperators. This field enables to implement the methods find that perform lookup for a name. Looking for a name can start from an initial scope at the root of our dictionary (see the method findAbsolute). It also can start from an initial scope defined by a NestedContext (see the methods find). If the name is not found, these methods return **nullptr**.

Invariants of the class GlobalContext

The following invariants reflect the organization inside the field \_logicTable:

* The def.\_parent field of a logic definition def of type NestedContext if not **nullptr** has def in its sons (see the virtual method NestedContext::ssons).
* Any son of a logic node context (see the method NestedContext::ssons) has a NestedContext::\_parent equal to context.
* A TemplateQualification node is always the son of a Qualification node. This Qualification node has QTEMPLATEINSTANCE as Qualification::tag.
* All the sons of a Qualification node with a QTEMPLATEINSTANCE as Qualification::tag are TemplateQualification.
* In an OverloadedFunctions / OverloadedLogicOperators has a logic\_info with a method flag, it should be inside a TemplateQualification or a Qualification corresponding to a **class**, a **struct** or a **union**. Note that instances of template functions correspond to a unique OverloadedFunctions.

Declaration of the class GlobalContext

**class** GlobalContext {

**private**:

**int** \_variableNumber;

**private**:

NestedContext::SoonsSet \_logicTable;

**void** init(); *// add Utf8\_logic::boolean*

**public**:

GlobalContext() : \_variableNumber(0) { init(); }

~GlobalContext();

**int**& variableNumber() { **return** \_variableNumber; }

**const** NestedContext::SoonsSet& logicTable() **const** { **return** \_logicTable; }

NestedContext::SoonsSet& logicTable() { **return** \_logicTable; }

NestedContext\* find(**const** std::string& identifier, NestedContext\* start) **const**;

NestedContext\* find(qualified\_name identifier, NestedContext\* start) **const**;

NestedContext\* findAbsolute(qualified\_name identifier) **const**;

};

The class Clang\_utils::VirtualDeclRegistration

The class VirtualDeclRegistration is a union class between Clang\_utils and VisitTable. It helps at discovering if a class or a type has been soon declared or soon defined in the context of translating a clang::QualType into a typ in the intermediate representation (see the methods Clang\_utils::makeType and Clang\_utils::makePlainType).

If the translation depends on a class name that has not been encountered like in the case

**template** <class T> List {

...

};

*// List<Integer> is visited here and nobody knows what is Integer at this point*

**class** Integer { ... };

List<Integer> ...;

then the caller can insert the name before the current translation

**class** Integer;

*// List<Integer> is visited here and nobody knows what is Integer at this point*

If the translation depends on a type definition that has not been encountered like in the case

**template** <class T> List {

...

};

*// List<Integer> is visited here and nobody knows what is Integer at this point*

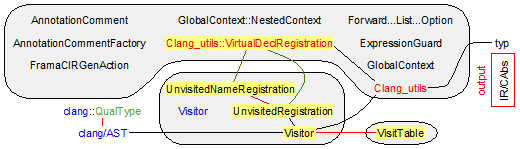
**typedef** … Integer;

List<Integer> ...;

then the caller should wait for the visit of the type definition before outputting the current translation

The class VirtualDeclRegistration is implemented by the classes ClangVisitor::UnvisitedRegistration and ClangVisitor::UnvisitedNameRegistration. When a clang declaration is visited, the method registerDecl is automatically called and depending on the status of this declaration – has been visited or not – registerDecl is likely to record the unvisited declarations for making them available to the methods VisitTable::VisitTable::setInstanceClassAsComplete, VisitTable::VisitTable::addIncompleteFunction.

The class environment of VirtualDeclRegistration is defined on the schema below:



Declaration of the class VirtualDeclRegistration

**class** VirtualDeclRegistration {

**private**:

**bool** \_doesRegisterDecl;

**protected**:

**void** setRegisterDecl() { \_doesRegisterDecl = **true**; }

**void** clearRegisterDecl() { \_doesRegisterDecl = **false**; }

**public**:

VirtualDeclRegistration() : \_doesRegisterDecl(**false**) {}

VirtualDeclRegistration(**const** VirtualDeclRegistration& source) = **default**;

**virtual** ~VirtualDeclRegistration() {}

**bool** doesRegisterDecl() **const** { **return** \_doesRegisterDecl; }

**virtual** **void** registerDecl(**const** clang::Decl\* decl) {}

**virtual** VirtualDeclRegistration\* getNameRegistration() { **return** **this**; }

};

The class Clang\_utils

The class Clang\_utils defines many utility methods to translate the C++ world into the world of the Intermediate Representation defined by the file “intermediate\_format.ast”. The unit ACSL++ and the unit ClangVisitor make intensive use of such translations.

The main services that our class provides are listed below:

* makeQualifiedName translates any named declaration (clang::NamedDecl) into a fully qualified named in the world of the Intermediate Representation.
* makeSignature translates any function signature (clang::FunctionDecl) into a signature in the world of the Intermediate Representation.
* makeType (and makePlainType) translates any C++ type (clang::QualType and clang::Type) into a type in the world of the Intermediate Representation in the C++ construction world.
* makeLogicType (and makePlainType) translates any C++ type (clang::Type) into a type in the world of the Intermediate Representation in the logic world.
* retrieveTypeOfField returns the offset in the logic world of ACSL++ of a field name in a record. This method takes care about inheritance.
* getTemplateExtension translates the instantiation of templates (functions, methods, classes) into the Intermediate Representation.

For that, the user of our class should give some information that is updated during the visit of the clang/AST.

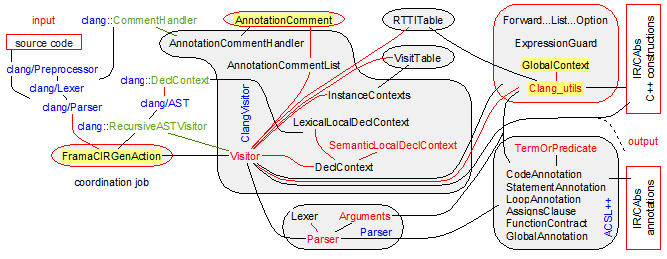
Initially he/she should provide the clang::ASTContext that is our entry point in the clang AST.

Then he/she is likely to provide the current scope as a clang::DeclContext. This is the case to activate the logical dictionary \_acslContext of type GlobalContext. This is also the case to give the meaning of a name through the method makeQualifiedName(**const** clang::DeclContext\* context, **const** **char**\* name, **const** clang::NamedDecl\* decl=**nullptr**, tkind\* templateParameters=**nullptr**).

The user has also to maintain the current template instantiation context that is simply a stack of instance arguments associated to their template classes/methods. The field \_defaultTemplateInstances is required, because during the visit, we are likely to encounter the template declaration without any instance arguments, these ones sometimes being implicit in the clang and/or the annotation context. In the generation of the Intermediate Representation translation, we have to explicit these arguments and find the right ones in our stack \_defaultTemplateInstances. For the user, the maintenance actions simply consist in calling the methods pushTemplateInstance / popTemplateInstance, each time we start / end the visit of an instance.

Our class internally manages all other information. It handles the anonymous declarations by giving them a unique name (see the fields \_anonymousIdent and \_anonymousMap) for the Intermediate Representation.

The class environment of Clang\_utils is defined on the schema below:



Fields of the class Clang\_utils

**mutable** **int** \_anonymousIdent;

To manage anonymous **namespace**, anonymous **struct**, anonymous **union**, anonymous fields, variables, functions, we provide them a unique identifier within the framework of the translation unit. As the lifetime of our Clang\_utils is also the translation unit lifetime, it is convenient to manage this identifier in our class.

The initial value of \_anonymousIdent is 0. Then, it is incremented each time a new clang declaration without any name is visited. The new incremented value helps to build a unique name starting with “anonymous\_ ”. At the same time, the map \_anonymousMap registers the mapping clang declaration → unique name. So as local invariant, we have \_anonymousIdent = \_anonymousMap.size().

**mutable** std::map<**const** clang::Decl\*, std::string> \_anonymousMap;

This field manages the registration of newly visited declarations that have no name. To manage the translation of such declarations, we give them a unique name thanks the field \_anonymousIdent. Our field \_anonymousMap only registers the mapping clang declaration → unique name to provide this name for latter visit of the same declaration. This map also enables to implement the methods findAnonymousName and findAnonymousDecl.

clang::ASTContext\* \_context;

This field is a reference to an entry point in the clang AST. It is useful to print the error messages in a user readable format.

**mutable** GlobalContext \_acslContext;

This field holds the logic functions, the logic variables, the logic types declared in some ACSL++ annotations and that have a global visibility. It is organized as a hierarchic dictionary whose nodes are declarations of namespaces, classes and instance of template classes and whose leaves are the logic definitions themselves. This field is in fact managed in the Parser::Parser::Arguments class that has a read/write access to it thanks to the property globalAcslContext – note that the lifetime of Parser::Parser::Arguments is limited to the parsing of the current of annotation and it is far less than the lifetime of our Clang\_utils. To retrieve logical definitions, it is necessary to provide an initial position in the hierarchy of the dictionary. The method queryDeclLogicScope supplies this position at the initialization of Parser::Parser::Arguments that has to provide the current annotation scope as a clang::DeclContext.

**mutable** std::vector<std::pair<**const** clang::Decl\*, **const** clang::TemplateArgumentList\*> > \_defaultTemplateInstances;

This field is a stack of template instance arguments that are updated by the visit of the clang AST by the methods pushTemplateInstance and popTemplateInstance. This field is required because some encountered clang nodes only refer to template declarations without any instance arguments; these ones are sometimes implicit in the clang and/or the annotation context. Hence \_defaultTemplateInstances enables to look for the right arguments associated the template clang::Decl – see the methods findInstanceArguments and findTemplateArgument.

**bool** \_annotError;

\_annotError is a global field that controls the error messages issued to the user. If \_annotError is set to **true**, the parsing (and so the generation in the intermediate format) stops on the first parsing error. In a C++ philosophy, \_annotError is rather set to **false**, because the parsing being relatively slow, the user wants to correct about one to ten errors before launching a new translation. In a OCaml (and/or logic) philosophy, \_annotError is rather set to **true**, because the parsing being relatively quick, the user then avoids to conduct reflections on the origin of the error – influenced by the previous error or a new error.

**bool** \_doesGenerateImplicitMethods;

This field is a global flag that manages the generation of implicit methods (default constructor, copy constructor, move constructor, default assignment, default move assignment, default destructor). If \_doesGenerateImplicitMethods = **true**, clang generates the body of these implicit methods (with consistent names with respect to the ones given by the OCaml plugin fclang). If \_doesGenerateImplicitMethods = **false**, the OCaml plugin fclang takes charge of the whole body generation.

**bool** \_isVerbose;

This field is a global flag that enables to follow the ACSL++ grammar rules that are used in the translation of the ACSL++ comments. If \_isVerbose is **false**, the ACSL++ grammar rules are not visible in the standard output.

Declaration of the class Clang\_utils

**class** Clang\_utils {

**private**:

**mutable** **int** \_anonymousIdent;

**mutable** std::map<**const** clang::Decl\*, std::string> \_anonymousMap;

clang::ASTContext\* \_context;

**mutable** GlobalContext \_acslContext;

**mutable** std::vector<std::pair<**const** clang::Decl\*, **const** clang::TemplateArgumentList\*> > \_defaultTemplateInstances;

**bool** \_annotError;

**bool** \_doesGenerateImplicitMethods;

**bool** \_isVerbose;

**bool** isSpecificType(**bool** (Clang\_utils::\*builtinMethod)(clang::BuiltinType **const**\*) **const**, **const** clang::Type\* type) **const**;

**public**:

Clang\_utils(clang::ASTContext\* ctxt) : \_anonymousIdent(), \_context(ctxt), \_annotError(), \_doesGenerateImplicitMethods(), \_isVerbose() { }

**void** setAnnotError() { \_annotError = **true**; }

**void** setGenerateImplicitMethods() { \_doesGenerateImplicitMethods = **true**; }

**void** setVerbose() { \_isVerbose = **true**; }

**bool** stopOnAnnotError () **const** { **return** \_annotError; }

**bool** doesGenerateImplicitMethods() **const** { **return** \_doesGenerateImplicitMethods; }

**bool** isVerbose() **const** { **return** \_isVerbose; }

**void** pushTemplateInstance(**const** clang::Decl\* templateDecl, **const** clang::TemplateArgumentList\* instanceArguments)

{ \_defaultTemplateInstances.push\_back(std::make\_pair(templateDecl, instanceArguments)); }

**void** popTemplateInstance(**const** clang::Decl\* templateDecl)

{ **assert**(\_defaultTemplateInstances.size() > 0 && \_defaultTemplateInstances.back().first == templateDecl); \_defaultTemplateInstances.pop\_back(); }

**bool** hasTemplateInstance() **const** { **return** \_defaultTemplateInstances.size() > 0; }

**const** clang::TemplateArgumentList\* findInstanceArguments(**const** clang::Decl\* templateDecl) **const**;

**const** clang::TemplateArgument\* findTemplateArgument(**const** std::string& name, **const** clang::NamedDecl\*& templateParameter) **const**;

GlobalContext::NestedContext\* queryDeclLogicScope(**const** clang::DeclContext\* clangScope) **const**;

**void** set\_context(clang::ASTContext\* ctxt) { \_context = ctxt; }

**const** clang::NamedDecl\* findAnonymousDecl(**const** std::string& name) **const**;

std::string findAnonymousName(**const** clang::NamedDecl\* decl) **const**;

qualified\_name makeQualifiedName(**const** clang::NamedDecl&) **const**;

signature makeSignature(**const** clang::FunctionDecl&) **const**;

qualified\_name makeQualifiedName(**const** clang::DeclContext\*Ctx, **const** **char**\* name, **const** clang::NamedDecl\* decl=**NULL**, tkind\* templateParameters=**NULL**) **const**;

GlobalContext& globalAcslContext() **const** { **return** \_acslContext; }

**bool** isIntegralType(clang::BuiltinType **const**\* typ) **const**

{ **return** (typ->getKind() >= clang::BuiltinType::Bool && typ->getKind() <= clang::BuiltinType::Int128); }

**bool** isSignedType(clang::BuiltinType **const**\* typ) **const**

{ **return** (typ->getKind() >= clang::BuiltinType::Char\_S && typ->getKind() <= clang::BuiltinType::Int128); }

**bool** isArithmeticType(clang::BuiltinType **const**\* typ) **const**

{ **return** (typ->getKind() >= clang::BuiltinType::Bool && typ->getKind() <= clang::BuiltinType::LongDouble); }

**bool** isFloatingType(clang::BuiltinType **const**\* typ) **const**

{ **return** (typ->getKind() >= clang::BuiltinType::Half && typ->getKind() <= clang::BuiltinType::LongDouble); }

typ makeBuiltinType(clang::BuiltinType **const**\* typ) **const**;

logic\_type makeBuiltinLogicType(clang::BuiltinType **const**\* typ) **const**;

**typedef** std::vector<**const** clang::Decl\*> UnvisitedDecls;

typ makePlainType(clang::QualType **const**& qt, VirtualDeclRegistration\* declRegistration=**NULL**) **const**;

qual\_type makeType(clang::QualType **const**& qt, VirtualDeclRegistration\* declRegistration=**NULL**) **const**;

logic\_type makeLogicType(clang::Type **const**\* type) **const**;

logic\_type makePointedType(clang::Type **const**\* type) **const**;

logic\_type makeReferencedType(clang::Type **const**\* type) **const**;

logic\_type makeElementArrayType(clang::Type **const**\* type) **const**;

logic\_type logicArithmeticPromotion(clang::Type **const**\* type) **const**;

**bool** isIntegralType(clang::Type **const**\* type) **const** { **return** isSpecificType(&Clang\_utils::isIntegralType, type); }

**bool** isSignedType(clang::Type **const**\* type) **const** { **return** isSpecificType(&Clang\_utils::isSignedType, type); }

**bool** isArithmeticType(clang::Type **const**\* type) **const** { **return** isSpecificType(&Clang\_utils::isArithmeticType, type); }

**bool** isFloatingType(clang::Type **const**\* type) **const** { **return** isSpecificType(&Clang\_utils::isFloatingType, type); }

**bool** isPointerType(clang::Type **const**\* type) **const**;

**bool** isReferenceType(clang::Type **const**\* type) **const**;

**bool** isArrayType(clang::Type **const**\* type) **const**;

**void** retrieveTypeOfField(clang::Type **const**\* type, **const** std::string& fieldName, term\_offset& offset, logic\_type& ltype, std::string& errorMessage, **const** clang::ASTContext\* clangAST, clang::Sema\* clangSema, **const** clang::SourceLocation& location, **const** RTTITable& rttiTable) **const**;

template\_parameter getTemplateExtension(**const** clang::TemplateArgument& parameter) **const**;

*/\* template\_parameter \*/* list getTemplateExtension(**const** clang::TemplateArgumentList& parameters) **const**;

**const** **char**\* get\_field\_name(**const** clang::NamedDecl\*) **const**;

**const** **char**\* get\_aggregate\_name(**const** clang::RecordDecl\*, tkind\* templateParameters) **const**;

list */\* specifier \*/* make\_specifier\_list(clang::QualType **const**& qt) **const**;

**int**& logicVariableNumber() **const** { **return** \_acslContext.variableNumber(); }

};

Methods of the class Clang\_utils

Private methods

**bool** isSpecificType(**bool** (Clang\_utils::\*builtinMethod)(clang::BuiltinType **const**\*) **const**, **const** clang::Type\* type) **const**;

Apply the builtinMethod on the builtin leaves of type and returns the result of this application under the condition that the nodes and the leaves of type have been successfully recognized. This method is a dispatch method. Depending on the type of type (see clang::Type::getTypeClass), this method recursively calls itself on the subtype of a node or it applies builtinMethod on the builtin type of a leaf type.

This method is exclusively called by the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType with their builtin method as argument. Hence it answers to these basic queries on clang::Type for the ACSL++ parser (see the classes Parser::Parser::Arguments and ACSL++::TermOrPredicate, ACSL++::TermOrPredicateList).

* + The method clang::Type::getTypeClass,
  + the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType,
  + the methods Parser::Parser::Arguments::isIntegralTypedefType, Parser::Parser::Arguments::isSignedTypedefType, Parser::Parser::Arguments::isArithmeticTypedefType, Parser::Parser::Arguments::isFloatingTypedefType,
  + the methods ACSL++::TermOrPredicate::isIntegralType, ACSL++::TermOrPredicate::isPlainBooleanType, ACSL++::TermOrPredicate::isSignedInteger, ACSL++::TermOrPredicate::isArithmeticType,
  + the logic type conversion methods ACSL++::TermOrPredicate::convertTermToPredicate, ACSL++::TermOrPredicate::typeBoolTerm , ACSL++::TermOrPredicate::makeCast, ACSL++::TermOrPredicate::logicArithmeticConversion, ACSL++::TermOrPredicate::conditionalConversion, ACSL++::TermOrPredicate::implicitConversion, ACSL++::TermOrPredicateList::updateCommonTypeWith,
  + the method ACSL++::TermOrPredicate::apply,
  + the methods isPointerType, isReferenceType, isArrayType.

Public methods

**const** clang::TemplateArgumentList\* findInstanceArguments(**const** clang::Decl\* templateDecl) **const**;

Find in the current template instance stack (see the field \_defaultTemplateInstances) the instance template arguments attached to the template declaration templateDecl. This method is required by the units ClangVisitor and Parser because during the clang/AST visit and the parsing of annotations, we are likely to encounter a template declaration without any instance arguments, these ones sometimes being implicit in the clang and/or the annotation context. These instance arguments should be in the current template instance stack. If our method fails to find templateDecl in our stack \_defaultTemplateInstances, it returns **nullptr**.

Our method is mainly called by makeQualifiedName when its clang::NamedDecl argument is a template class declaration. It is often the case during the parsing of ACSL++ annotations in instance classes but it also may occur during the visit of the clang/AST.

1. These instance arguments should be in the current template instance stack.
2. Our method should not return **nullptr** else a parse error should be issued by the caller.
   * The field \_defaultTemplateInstances,
   * the methods pushTemplateInstance, popTemplateInstance,,
   * the method makeQualifiedName,
   * the method findTemplateArgument.

**const** clang::TemplateArgument\* findTemplateArgument(**const** std::string& name, **const** clang::NamedDecl\*& templateParameter) **const**;

Find in the current template instance stack (see the field \_defaultTemplateInstances) the instance template argument attached to name. This method is required by the unit Parser because during the clang/AST visit and the parsing of annotations, we are likely to encounter a template argument name. The objective of our method is to return the instance argument corresponding to name. On the example below with name = “N”,

**template** <**typename** T, **unsigned** **long** N>

**class** Stack {

**public**:

**typedef** **unsigned** **long** size\_type;

**typedef** T value\_type;

*/\*@* ***logic******integer*** *Size() = sz;*

***logic******integer*** *Capacity() = N;*

***logic*** *T Top() = rep.elems[sz-1];*

***predicate*** *operator==(Stack s) = Size() == s.Size()*

*&& (****\forall*** *integer i; 0 <= i < Size() ==> rep.elems[i] == s.rep.elems[i]);*

***class******invariant*** *StackInvariant = sz <= N;*

***predicate*** *Empty() = Size() == 0;*

***predicate*** *Full() = Size() == Capacity();*

*\*/*

...

**private**:

array<T, N> rep;

size\_type sz;

};

Stack<int, 6> s;

our method returns the clang::TemplateArgument\* attached the instance 6. templateParameter is then the formal template parameter N. If our method fails to find name in our stack \_defaultTemplateInstances, it returns **nullptr**.

Our method is called by makeLogicType when it encounters a clang::TemplateTypeParmType (see the classes ACSL++::LogicType, ACSL++::GlobalAnnotation, ACSL++::TermOrPredicate).

It is also called by Parser::Parser::Arguments::isCodeName to determine the kind of identifier in argument – see also the parse methods ACSL++::LogicType::readToken and ACSL++::TermOrPredicate::readToken.

* + The field \_defaultTemplateInstances,
  + the methods pushTemplateInstance, popTemplateInstance,,
  + the methods makeLogicType and Parser::Parser::Arguments::isCodeName,
  + the classes ACSL++::LogicType, ACSL++::GlobalAnnotation, ACSL++::TermOrPredicate,
  + the method findInstanceArguments.

GlobalContext::NestedContext\* queryDeclLogicScope(**const** clang::DeclContext\* clangScope) **const**;

Return the current position in the hierarchic dictionary \_acslContext of global logic definitions. The hierarchic levels in this dictionary (seen the class GlobalContext) are namespaces, classes and instances of namespace classes. This is the same for clangScope where the semantic links clang::DeclContext::getParent provides namespaces, classes and instances of namespace classes.

The returned GlobalContext::NestedContext gives a natural scope to look for identifiers that are logic entities. This result will be in concurrence with the clangScope to identify the identifiers used in ACSL++. It also defines the namespace and class scope enabling to store the logic constructs at the right places. This method is exclusively called by the constructor Parser::Parser::Arguments::Arguments at each visit of an ACSL++ annotation.

* + The field \_acslContext, the method clang::DeclContext::getParent and the classes GlobalContext, NestedContext, Qualification, TemplateQualification,
  + the constructor Parser::Parser::Arguments::Arguments.

**const** clang::NamedDecl\* findAnonymousDecl(**const** std::string& name) **const**;

Return the clang declaration that is associated to name in the format of the Intermediate Representation. With respect to the direct method fiindAnonymousName, this is its reverse function that is a bit costly since it iterates on the map \_anonymousMap until it finds a pair (result, name) and then it returns result. This method helps to implements some helper methods of Parser::Parser::Arguments that have some questions about the result of makeQualifiedName: is it an integral type (see the method Parser::Parser::Arguments::isIntegratTypedefType), is it a pointer type (see the method Parser::Parser::Arguments::isArrayTypedefType)? This method could be avoided since the argument of makeQualifiedName (a clang::NamedDecl) should be the result of our method.

* + The field \_anonymousMap,
  + the methods findAnonymousName, get\_field\_name, get\_aggregate\_name,
  + the method makeQualifiedName,
  + the methods Parser::Parser::Arguments::isIntegralTypedefType, Parser::Parser::Arguments::isSignedTypedefType, Parser::Parser::Arguments::isArithmeticTypedefType, Parser::Parser::Arguments::isFloatingTypedefType, Parser::Parser::Arguments::isPointerTypedefType, Parser::Parser::Arguments::isReferenceTypedefType, Parser::Parser::Arguments::isArrayTypedefType, Parser::Parser::Arguments::makeTypeOfPointed, Parser::Parser::Arguments::makeTypeOfReferenced, Parser::Parser::Arguments::makeTypeOfArrayElement.

std::string findAnonymousName(**const** clang::NamedDecl\* decl) **const**;

Return the identifier in the format of the Intermediate Representation of the anonymous declaration decl. The declaration of the **namespace**, **struct**, **union**, field should have been previously visited. Hence decl is a valid key of the map \_anonymousMap.

This method is mainly called to handle the translation of clang::IndirectFieldDecl: an instance of this class is created to represent a field injected from an anonymous union/struct into the parent scope.

1. decl is a valid key of the map \_anonymousMap. That means that one of the methods makeQualifiedName, get\_field\_name, get\_aggregate\_name has been called with decl.
   * The field \_anonymousMap,
   * the methods findAnonymousDecl, get\_field\_name, get\_aggregate\_name,
   * the class clang::IndirectFieldDecl and the method ACSL++::TermOrPredicate::readToken,
   * the methods makeQualifiedName, get\_field\_name, get\_aggregate\_name.

**const** **char**\* get\_field\_name(**const** clang::NamedDecl\* decl) **const**;

Return the identifier in the format of the Intermediate Representation corresponding to the field decl. If decl is not an anonymous field it returns a copy of decl->getName(). If decl is an anonymous field the implementation looks at decl in the map \_anonymousMap. If it finds it, it returns a copy of the uniquely associated name else it creates a fresh identifier starting with anonymous\_ and it returns this identifier.

This method is mainly called to handle the translation of fields of type clang::VarDecl, clang::FieldDecl and the translation of clang::MemberExprClassStmt and of clang::CXXCtorInitializer verifying clang::CXXCtorInitializer::isMemberInitializer.

1. decl is a clang field declaration.
   * The field \_anonymousMap,
   * the methods ClangVisitor::Visitor::VisitVarDecl, ClangVisitor::Visitor::VisitFieldDecl, ClangVisitor::Visitor::makeExpression, ClangVisitor::Visitor::insertConstructorPreambleIn,
   * the methods findAnonymousName, get\_aggregate\_name,
   * the class clang::IndirectFieldDecl and the method ACSL++::TermOrPredicate::readToken,
   * the methods makeQualifiedName, get\_field\_name, get\_aggregate\_name.

**const** **char**\* get\_aggregate\_name(**const** clang::RecordDecl\* decl, tkind\* templateParameters) **const**;

Return the identifier in the format of the Intermediate Representation corresponding to the record decl. If decl is a template instance class then templateParameters receives the result of tkind\_TTemplateInstance(getTemplateExtension(…)) applied on the instantiation argument list. If decl is an anonymous **struct** or **class** the implementation looks at decl in the map \_anonymousMap. If it finds it, it returns a copy of the uniquely associated name else it creates a fresh identifier starting with anonymous\_ and it returns this identifier.

This method is called by makePlainType makeLogicType, to handle the translation of clang clang::RecordDecl.

* + The field \_anonymousMap,
  + the methods makePlainType makeLogicType, ClangVisitor::Visitor::VisitRecordDecl, ClangVisitor::Visitor::insertNamedDeclaration,
  + the methods findAnonymousName, get\_field\_name,
  + the class clang::IndirectFieldDecl and the method ACSL++::TermOrPredicate::readToken,
  + the methods makeQualifiedName, get\_field\_name, get\_aggregate\_name.

qualified\_name makeQualifiedName(**const** clang::NamedDecl& decl) **const**;

Return the name corresponding to decl in the Intermediate Representation. The implementation takes anonymous decl into account and calls makeQualifiedName(**const** clang::DeclContext\*, **const** **char**\*, **const** clang::NamedDecl\*, tkind\*) for the complete implementation.

The units ClangVisitor, VisitTable, TableRTTI (and also ACSL++) use this important method.

* + The field \_anonymousMap,
  + the method makeQualifiedName(**const** clang::DeclContext\*, **const** **char**\*, **const** clang::NamedDecl\*, tkind\*),
  + the method makeSignature.

signature makeSignature(**const** clang::FunctionDecl& decl) **const**;

Return the signature corresponding to decl in the Intermediate Representation. The implementation calls makeType for the result and the type of the arguments.

* + The method makeType,
  + the methods makeQualifiedName.

qualified\_name makeQualifiedName(**const** clang::DeclContext\*Ctx, **const** **char**\* name, **const** clang::NamedDecl\* decl=**NULL**, tkind\* templateParameters=**NULL**) **const**;

Return in the Intermediate Representation the name corresponding to name (or decl if name = **nullptr**) in the semantic context Ctx. The implementation takes anonymous Ctx into account. Ctx (and its hierarchy through the link clang::DeclContext::getParent) enables to compute the part \_qualified\_name::prequalification of the result. name (or decl if name = **nullptr**) corresponds to the part \_qualified\_name::decl\_name of the result.

The units ClangVisitor, VisitTable, TableRTTI (and also ACSL++) use this important method.

* + The field \_anonymousMap,
  + the method makeQualifiedName(**const** clang::NamedDecl&),
  + the method makeSignature.

typ makeBuiltinType(clang::BuiltinType **const**\* typ) **const**;

Depending on typ->getKind(), this method return the type corresponding to typ in the Intermediate Representation at the level of the C++ constructions.

This method is called by the methods makePlainType and makeType that perform a high level translation of clang::Type and clang::QualType.

* + The methods makePlainType and makeType,
  + the method makeBuiltinLogicType.

logic\_type makeBuiltinLogicType(clang::BuiltinType **const**\* typ) **const**;

Depending on typ->getKind(), this method return the type corresponding to typ in the Intermediate Representation at the level of the ACSL++ annotations.

This method is called by the methods makeLogicType and logicArithmeticPromotion that perform a high level translation of clang::Type and clang::QualType.

* + The methods makeLogicType and logicArithmeticPromotion,
  + the method makeBuiltinType.

typ makePlainType(clang::QualType **const**& qt, VirtualDeclRegistration\* declRegistration=**NULL**) **const**;

Depending on qt.getTypePtr()->getTypeClass() this recursive method translates the clang::Type associated to qt into a typ in the Intermediate Representation at the level of C++ constructions. The implementation calls the methods makeBuiltinType, makeQualifiedName for the **struct**, **class**, **union**, **enum**. Note that **typedef** are inlined for the makePlainType method and not for makeLogicType since a **typedef** can carry specific ACSL++ annotations.

This method is called by the method makeType and is intensively used in the ClangVisitor unit..

* + The methods makeBuiltinType, makeQualifiedName,
  + the method makeType,
  + the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isReferenceType, isArrayType,
  + the methods makeLogicType, logicArithmeticPromotion, makePointedType, makeReferencedType, makeElementArrayType.

qual\_type makeType(clang::QualType **const**& qt, VirtualDeclRegistration\* declRegistration=**NULL**) **const**;

Depending on qt (see the methods clang::QualType::isLocalRestrictQualified, clang::QualType::isLocalVolatileQualified, clang::QualType::isLocalConstQualified) and on qt.getTypePtr()->getTypeClass() this recursive method translates the clang::Type associated to qt into a typ in the Intermediate Representation at the level of C++ constructions. The implementation calls the methods makePlainType and make\_specifier\_list for the **const**, **restrict**, **volatile**. Note that **typedef** are inlined for the makeType method and not for makeLogicType since a **typedef** can carry specific ACSL++ annotations.

This method is intensively used in the ClangVisitor unit.

* + The methods make\_specifier\_list, makePlainType, makeBuiltinType, makeQualifiedName,
  + the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isReferenceType, isArrayType, isPointerType, isReferenceType, isArrayType,
  + the methods makeLogicType, logicArithmeticPromotion, makePointedType, makeReferencedType, makeElementArrayType.

logic\_type makeLogicType(clang::Type **const**\* type) **const**;

Depending on qt.getTypePtr()->getTypeClass() this recursive method translates the clang::Type associated to qt into a logic\_type in the Intermediate Representation at the level of ACSL++ annotations. The implementation calls the methods makeBuiltinLogicType, makeQualifiedName with the result of get\_aggregate\_name for the **struct**, **class**, **union**, and makeQualifiedName for **enum**, **typedef**. Note that **typedef** are inlined for the makeType method and not for makeLogicType since a **typedef** can carry specific ACSL++ annotations.

This method is called by the methods makePointedType, makeReferencedType, makeElementArrayType, retrieveTypeOfField and in Parser::Parser::Arguments::makeLogicType, Parser::Parser::Arguments::createResultType is intensively used in the ACSL++ unit.

* + The methods makeBuiltinLogicType, makeQualifiedName, get\_aggregate\_name,
  + the methods makePointedType, makeReferencedType, makeElementArrayType, retrieveTypeOfField,
  + the methods Parser::Parser::Arguments::makeLogicType, Parser::Parser::Arguments::createResultType,
  + the method logicArithmeticPromotion, isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isReferenceType, isArrayType,
  + the methods makeType, makePlainType.

logic\_type makePointedType(clang::Type **const**\* type) **const**;

Depending on type->getTypeClass() this recursive method first recognizes in type a pointer on a subtype that is a clang::Type. Then it translates subtype into a logic\_type in the Intermediate Representation at the level of ACSL++ annotations, usually by calling the method makeLogicType.

This method is called by the method Parser::Parser::Arguments::makeTypeOfPointed and is used in ACSL++::TermOrPredicate.

1. The method isPointerType should have returned **true** on type. This means that type is semantically equal to subtype\*.
   * The methods makeLogicType,
   * the method isPointerType,
   * the methods Parser::Parser::Arguments::makeTypeOfPointed, ACSL++::TermOrPredicate::typeOfPointed, ACSL++::TermOrPredicate::typeOfArrayElement, ACSL++::TermOrPredicate::makeMemoryShift, ACSL++::TermOrPredicate::conditionalConversion involving pointers, ACSL++::TermOrPredicate::implicitConversion involving conversions from arrays to pointers, ACSL++::TermOrPredicate::apply for the logic interpretation of operator ->, operator[], comparison of pointers and arrays,
   * the method makeType,
   * the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isReferenceType, isArrayType,
   * the methods makeLogicType, logicArithmeticPromotion, makeReferencedType, makeElementArrayType.

logic\_type makeReferencedType(clang::Type **const**\* type) **const**;

Depending on type->getTypeClass() this recursive method first recognizes in type a reference on a subtype that is a clang::Type. Then it translates subtype into a logic\_type in the Intermediate Representation at the level of ACSL++ annotations, usually by calling the method makeLogicType.

This method is called by the method Parser::Parser::Arguments::makeTypeOfReferenced and is used in ACSL++::TermOrPredicate.

1. The method isReferenceType should have returned **true** on type. This means that type is semantically equal to subtype&.
   * The methods makeLogicType,
   * the method isReferenceType,
   * the methods Parser::Parser::Arguments::makeTypeOfReferenced, ACSL++::TermOrPredicate::typeOfPointed, ACSL++::TermOrPredicate::makeMemoryShift, ACSL++::TermOrPredicate::apply for the logic interpretation of operator .,
   * the method makeType,
   * the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isArrayType,
   * the methods makeLogicType, logicArithmeticPromotion, makePointedType, makeElementArrayType.

logic\_type makeElementArrayType(clang::Type **const**\* type) **const**;

Depending on type->getTypeClass() this recursive method first recognizes in type an array on a subtype that is a clang::Type. Then it translates subtype into a logic\_type in the Intermediate Representation at the level of ACSL++ annotations, usually by calling the method makeLogicType.

This method is called by the method Parser::Parser::Arguments::makeTypeOfArrayElement and is used in ACSL++::TermOrPredicate. Note that the method ACSL++::TermOrPredicate::typeOfArrayElement rather calls the method makePointedType which is more general.

1. The method isPointerType should have returned **true** on type. This means that type is semantically equal to subtype\*.
   * The methods makeLogicType,
   * the method isArrayType,
   * the methods Parser::Parser::Arguments::makeTypeOfArrayElement, ACSL++::TermOrPredicate::typeOfPointed, ACSL++::TermOrPredicate::typeOfArrayElement, ACSL++::TermOrPredicate::makeMemoryShift, ACSL++::TermOrPredicate::implicitConversion involving conversions from arrays to pointers, ACSL++::TermOrPredicate::apply for the logic interpretation of operator[], comparison of pointers and arrays,
   * the method makeType,
   * the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isReferenceType,
   * the methods makeLogicType, logicArithmeticPromotion, makePointerType, makeReferencedType.

logic\_type logicArithmeticPromotion(clang::Type **const**\* type) **const**;

Depending on qt.getTypePtr()->getTypeClass() this recursive method translates the clang::Type associated to qt into a logic\_type in the Intermediate Representation that is an arithmetic type at the level of ACSL++ annotations. The implementation filters the implementation of makeLogicType with arithmetic types. The implementation calls the methods makeBuiltinLogicType for builtin types. Note that **typedef** are inlined since the result is mainly used for implicit conversions.

This method is called by the methods Parser::Parser::Arguments::logicArithmeticPromotion and is intensively used in the ACSL++::TermOrPredicate class.

1. The method isArithmeticType should have returned **true** on type.
   * The methods makeBuiltinLogicType,
   * the method makeLogicType, isArithmeticType,
   * the methods Parser::Parser::Arguments::logicArithmeticPromotion, ACSL++::TermOrPredicate::logicArithmeticPromotion, ACSL++::TermOrPredicate::carithmeticConversion, ACSL++::TermOrPredicate::apply for the operator-, operator~,
   * the method isIntegralType, isSignedType, isFloatingType, isPointerType, isReferenceType, isArrayType,
   * the methods makeType, makePlainType.

**bool** isPointerType(clang::Type **const**\* type) **const**;

Depending on type->getTypeClass() this recursive method returns **true** if and only if it recognizes in type a pointer on a subtype that is a clang::Type. If it returns **true**, this means that type is semantically equal to subtype\* and the method makePointedType can be safely called on type.

This method is called by the method Parser::Parser::Arguments::isPointerTypedefType and is used in ACSL++::TermOrPredicate.

* + the method makePointedType,
  + the methods Parser::Parser::Arguments::isPointerTypedefType, ACSL++::TermOrPredicate::isCPointerType, ACSL++::TermOrPredicate::isPointerType, ACSL++::TermOrPredicate::isPlainPointerType, ACSL++::TermOrPredicate::isPointerCharType, ACSL++::TermOrPredicate::makeCast, ACSL++::TermOrPredicate::applyTermCast, ACSL++::TermOrPredicate::typeBoolTerm, ACSL++::TermOrPredicate::conditionalConversion, ACSL++::TermOrPredicate::implicitConversion, ACSL++::TermOrPredicate::TermOrPredicateList::updateCommonTypeWith,
  + the methods ACSL++::TermOrPredicate::typeOfPointed, ACSL++::TermOrPredicate::typeOfArrayElement, ACSL++::TermOrPredicate::makeMemoryShift,
  + the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isReferenceType, isArrayType,
  + the methods makeLogicType, logicArithmeticPromotion, makeReferencedType, makeElementArrayType.

**bool** isReferenceType(clang::Type **const**\* type) **const**;

Depending on type->getTypeClass() this recursive method returns **true** if and only if it recognizes in type a reference on a subtype that is a clang::Type. If it returns **true**, this means that type is semantically equal to subtype& and the method makeReferenceType can be safely called on type.

This method is called by the method Parser::Parser::Arguments::isReferenceTypedefType and is used in ACSL++::TermOrPredicate.

* + the method makeReferenceType,
  + the methods Parser::Parser::Arguments::isReferenceTypedefType, ACSL++::TermOrPredicate::isCReferenceType,
  + the methods ACSL++::TermOrPredicate::typeOfPointed, ACSL++::TermOrPredicate::makeMemoryShift,
  + the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isArrayType,
  + the methods makeLogicType, logicArithmeticPromotion, makePointedType, makeElementArrayType.

**bool** isArrayType(clang::Type **const**\* type) **const**;

Depending on type->getTypeClass() this recursive method returns **true** if and only if it recognizes in type an array on a subtype that is a clang::Type. If it returns **true**, this means that type is semantically equal to subtype[] and the method makeElementArrayType can be safely called on type.

This method is called by the method Parser::Parser::Arguments::isArrayTypedefType and is used in ACSL++::TermOrPredicate.

* + the method makeElementArrayType,
  + the methods Parser::Parser::Arguments::isArrayTypedefType, ACSL++::TermOrPredicate::isArrayType, ACSL++::TermOrPredicate::isCArrayType, ACSL++::TermOrPredicate::isPlainArrayType, ACSL++::TermOrPredicate::makeShift, ACSL++::TermOrPredicate::conditionalConversion, ACSL++::TermOrPredicate::apply, ACSL++::TermOrPredicate::readToken,
  + the methods ACSL++::TermOrPredicate::typeOfPointed, ACSL++::TermOrPredicate::typeOfArrayElement, ACSL++::TermOrPredicate::makeMemoryShift,
  + the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isReferenceType,
  + the methods makeLogicType, logicArithmeticPromotion, makePointedType, makeReferencedType.

**void** retrieveTypeOfField(clang::Type **const**\* type, **const** std::string& fieldName, term\_offset& offset, logic\_type& ltype, std::string& errorMessage, **const** clang::ASTContext\* clangAST, clang::Sema\* clangSema, **const** clang::SourceLocation& location, **const** RTTITable& rttiTable) **const**;

Depending on type->getTypeClass() this recursive method first recognizes in type a **class** T, a **struct** T or a **union** U. Then it looks for fieldname in the fields of T, U and in the inherited fields of T. When it founds the field:

* it converts the field access into an offset (see the result parameter offset)
* and it returns in the result parameter ltype the translation of the field clang type in the Intermediate Representation of logic types (see the method makeLogicType for the translation)

To find the field, we use clangSema->LookupQualifiedName with a search kind defined by clang::Sema::LookupMemberName and location. If the found field is an inherited one, the method rttiTable.getInheritancePath(…) enables to define the actual offset. If no field is found, an error message is appended into errorMessage.

This method is called by the method Parser::Parser::Arguments::retrieveTypeOfField and is used in ACSL++::TermOrPredicate.

1. type is semantically a **class** type, a **struct** type or a **union** type.
   * The methods makeLogicType, clang::Sema::LookupQualifiedName, RTTITable::getInheritancePath,
   * the methods Parser::Parser::Arguments::retrieveTypeOfField, ACSL++::TermOrPredicate::retrieveTypeOfField, ACSL++::TermOrPredicate::apply for the logic interpretation of operator., operator->,
   * the method makeType,
   * the methods isIntegralType, isSignedType, isArithmeticType, isFloatingType, isPointerType, isReferenceType, isArrayType,
   * the methods makeLogicType, logicArithmeticPromotion, makePointerType, makeReferencedType, makeArrayType,
   * the method get\_field\_name.

template\_parameter getTemplateExtension(**const** clang::TemplateArgument& argument) **const**;

Depending on argument.getKind(), this method translates the template instance argument argument into a template\_parameter to be used in the Intermediate Representation. The implementation handles the compilation constants and calls the methods makeType for the translation of types and makeQualifiedName for the translation of declarations.

This method is mainly called by the method getTemplateExtension(**const** clang::TemplateArgumentList&) **const** to translate a template instantiation list. Note that the function template\_parameter\_equal will be used as a match function. It is then very useful to differentiate the template instances at various level: in the ClangVisitor unit, but also in the Parser, ACSL++, RTTITable units.

* + The methods makeType and makeQualifiedName,
  + the method getTemplateExtension(**const** clang::TemplateArgumentList&) **const**,
  + the function template\_parameter\_equal.

*/\* template\_parameter \*/* list getTemplateExtension(**const** clang::TemplateArgumentList& parameters) **const**;

Translate a complete template instantiation list into a template\_parameter list that is valid in the Intermediate Representation of the C++ constructions and in the Intermediate Representation of ACSL++ annotations. The implementation simply calls the method getTemplateExtension(**const** clang::TemplateArgument&) on each argument of the list parameters.

This method is called by any unit that needs to translate instance classes. It concerns the units ClangVisitor, but also Parser, ACSL++, RTTITable.

* + The method getTemplateExtension(**const** clang::TemplateArgument&),
  + the methods Parser::Parser::Arguments::queryThisType, Parser::Parser::Arguments::queryThisLogicType, ClangVisitor::Visitor::makeTemporaryObjectExpression, ClangVisitor::Visitor::makeConstructExpression, ClangVisitor::Visitor::makeNewExpression, ClangVisitor::Visitor::makeCallExpression, ClangVisitor::Visitor::makeMemberCallExpression, ClangVisitor::Visitor::makeBaseToDerivedPointerCastExpression, ClangVisitor::Visitor::makeBaseToDerivedReferenceCastExpression, ClangVisitor::Visitor::makeDerivedToBasePointerCastExpression, ClangVisitor::Visitor::makeDerivedToBaseReferenceCastExpression, ClangVisitor::Visitor::makeDeclRefExpression, ClangVisitor::Visitor::makeImplicitValueInitExpression, ClangVisitor::Visitor::VisitFunctionDecl, ClangVisitor::Visitor::makeInheritanceList, RTTITable::DelayedMethodCalls::MethodCall::apply, RTTITable::RTTITable::addPvmtSetter, RTTITable::RTTITable::insertStaticVMTDefinition, RTTITable::RTTITable::insertStaticBaseClassesDefinition, RTTITable::RTTITable::retrieveMethodIndex.

The classes AnnotationComment and AnnotationCommentFactory

The AnnotationComment is an abstract class to make the parsing of the annotations independent from the ClangVisitor strategy. The class Parser::ACSLComment implements it. Hence ClangVisitor registers the AnnotationComment as valid annotations, but the ACSL Parser actually handles each annotation as a concrete Parser::ACSLComment. The class AnnotationCommentFactory completes the construction of this Factory Method design pattern.

An AnnotationComment object is registered each time the preprocessor encounters a comment that is valid (see the method AnnotationComment::isValid) as an annotation comment. This comment is then visited by the ClangVisitor::Visitor and it is parsed at that time to produce valid annotations in the Intermediate Representation defined by the file “intermediate\_format.ast”.

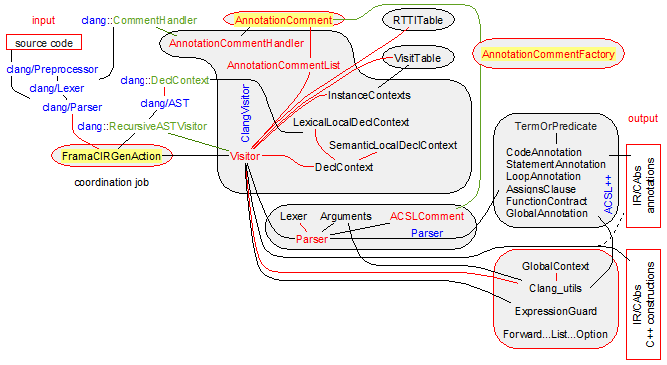
For that, clang provides a method clang::Preprocessor::addCommentHandler to register an instance of ClangVisitor::AnnotationCommentHandler that carries a ACSL++ comment detection mechanism. This ClangVisitor::AnnotationCommentHandler then stores it into ClangVisitor::AnnotationCommentList and attaches each comment it recognizes (see step/guideline 2 of the introduction of FramaCIRGenAction and the method initFrom)

* either to the global context (\_kind = KGlobal),
* or to the next declaration context (\_kind = KNext, KNextStatement, KNextLoop),
* or between two instructions context (\_kind = KLabel),
* or to the outer loop context (\_kind = KOuterLoop),

depending on the first words of the annotation. The location \_range enables to precisely locate the annotation.

When the visitor ClangVisitor::Visitor visits a declaration or an instruction (see step/guideline 3), it looks if there are annotations between this declaration and the location of the previous declaration. If this is the case, then the declaration is parsed with the adequate virtual method among parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract. Note that all of the implementations of these methods in Parser::ACSLComment call the Parser::Parser::parse method.

The class environment of AnnotationComment and AnnotationCommentFactory is defined on the schema below:



Fields of the class AnnotationComment

clang::SourceRange \_range;

Precise location of the comment. This field is set up by the constructor of our class.

Kind \_kind;

Attachment information for the comment. This field is set up by the initFrom method. This field is different from KUndefined for a comment that is actually a ACSL++ annotation. initFrom is called in step/guideline 2 of the introduction of FramaCIRGenAction. If \_kind ≠ KUndefined, the visitor ClangVisitor::Visitor will handle our annotation comment in step/guideline 3 and depending from \_kind, it will call:

* parseGlobal if \_kind = KGlobal,
* parseCodeAnnotation if \_kind = KLabel,
* parseLoopAnnotation if \_kind = KNextLoop,
* parseStatementAnnotation if \_kind = KNext or if \_kind = KNextStatement and if \_range is inside a function body,
* parseFunctionContract if \_kind = KNext and if \_range is before a function/method.

std::string\* \_commentText;

Content of the comment. This field is set up by the initFrom method. This field is different from **nullptr** for a comment that is actually a ACSL++ annotation.

**bool** \_isLineComment;

This field is **true** if and only if our comment starts with //. It is **false** if our comment is a standard C comment delimited by /\* \*/.

Declaration of the classes AnnotationComment and AnnotationCommentFactory

**class** AnnotationComment {

**public**:

**enum** Kind { KUndefined, KGlobal, KNext, KNextStatement, KNextLoop, KLabel, KOuterLoop };

**private**:

clang::SourceRange \_range;

Kind \_kind;

std::string\* \_commentText;

**bool** \_isLineComment;

**public**:

AnnotationComment(**const** clang::SourceRange& range) : \_range(range), \_kind(KUndefined), \_commentText(**NULL**), \_isLineComment(**false**) {}

AnnotationComment(**const** AnnotationComment& source)

: \_range(source.\_range), \_kind(source.\_kind), \_commentText(source.\_commentText), \_isLineComment(source.\_isLineComment) {}

**virtual** ~AnnotationComment() {}

**void** initFrom(**const** clang::SourceManager& sourceMgr);

**void** freeContent() { **if** (\_commentText) { **delete** \_commentText; \_commentText = **NULL**; } }

**const** std::string& getContent() { assert(\_commentText); **return** \*\_commentText; }

Kind getKind() **const** { **return** \_kind; }

**const** clang::SourceRange& getSourceRange() **const** { **return** \_range; }

clang::SourceLocation getSourceLocation() **const** { **return** \_range.getBegin(); }

**bool** isValid() **const** { **return** \_kind != KUndefined; }

**bool** isGlobal() **const** { **return** \_kind == KGlobal; }

**bool** isNext() **const** { **return** \_kind >= KNext && \_kind <= KNextLoop; }

**bool** isNextContract() **const** { **return** \_kind == KNext; }

**bool** isNextLoop() **const** { **return** \_kind == KNextLoop; }

**bool** isNextStatement() **const** { **return** \_kind == KNext; }

**bool** isLabel() **const** { **return** \_kind == KLabel; }

**bool** isOuterLoop() **const** { **return** \_kind == KOuterLoop; }

**bool** isLineComment() **const** { **return** \_isLineComment; }

**virtual** **void** parseGlobal(ForwardReferenceList& globals, ForwardReferenceList\* classContent, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc) {}

**virtual** **void** parseCodeAnnotation(ForwardReferenceList& codeContainer, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc) {}

**virtual** */\* code\_annotation \*/* list parseLoopAnnotation(**const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc) { **return** **NULL**; }

**virtual** **void** parseStatementAnnotation(ForwardReferenceList& codeContainer, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc) {}

**virtual** **void** parseFunctionContract(option& */\* function\_contract \*/* contract, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc) {}

**virtual** Kind getAnnotationKind(**const** **char**\*& start, size\_t length) **const** { **return** KUndefined; }

};

**class** AnnotationCommentFactory {

**public**:

**static** AnnotationComment\* createAnnotationComment(**const** clang::SourceManager& sourceMgr, **const** clang::SourceRange& range);

};

Methods of the class AnnotationComment

Public methods

**void** initFrom(**const** clang::SourceManager& sourceMgr);

Recognize the comment defined by \_range in the source code. sourceMgr helps to obtain the text of the comment from \_range. This recognition occurs during the step/guideline 2 of the introduction of FramaCIRGenAction before registering valid annotation comments into ClangVisitor::AnnotationCommentList.

The recognition process fills the fields \_isLineComment, \_kind, \_commentText with their final value. To define \_kind, it is necessary to parse the text of the comment. This is done by calling the virtual method getAnnotationKind that is implemented in derived classes (ex in Parser::ACSLComment). For valid annotation comments, getAnnotationKind should return an attachment different from KUndefined.

Note that our method initFrom is distinct from the constructor because it calls the virtual method getAnnotationKind.

* + The methods getAnnotationComment and Parser::ACSLComment::getAnnotationComment,
  + the fields \_isLineComment, \_kind, \_commentText,
  + the methods parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract.

**virtual** **void** parseGlobal(ForwardReferenceList& globals, ForwardReferenceList\* classContent, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc);

Parse the global annotation \_commentText in the semantic context defined by clangContext/scope and insert the Intermediate Representation translation into globals. This method is implemented by Parser::ACSLComment::parseGlobal for the ACSL++ logic …, predicate …, global …, inductive …, axiomatic …, model …, class …, type …, volatile … constructions.

This method is called by ClangVisitor::Visitor::parseGlobalComment before the visit of any declaration and at the end of the visit of the translation unit.

1. \_kind, the result of getAnnotationKind is KGlobal.
   * The method Parser::ACSLComment::parseGlobal,
   * the method getAnnotationComment and the fields \_kind, \_commentText,
   * the method ClangVisitor::Visitor::parseGlobalComment,
   * the methods parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract.

**virtual** **void** parseCodeAnnotation(ForwardReferenceList& codeContainer, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc);

Parse the code annotation \_commentText in the semantic context defined by clangContext/scope and insert the Intermediate Representation translation into codeContainer. This method is implemented by Parser::ACSLComment::parseCodeAnnotation for the ACSL++ assert construction.

This method is called by ClangVisitor::Visitor::parseLoopAnnotation before the visit of any loop (clang::DoStmt, clang::ForStmt, clang::WhileStmt) – see the methods ClangVisitor::Visitor::readStatementCommentUntil.

1. \_kind, the result of getAnnotationKind is KLabel.
   * The method Parser::ACSLComment::parseCodeAnnotation,
   * the method getAnnotationComment and the fields \_kind, \_commentText,
   * the method ClangVisitor::Visitor::parseCodeAnnotation, ClangVisitor::Visitor::readStatementCommentUntil,
   * the methods parseGlobal, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract.

**virtual** */\* code\_annotation \*/* list parseLoopAnnotation(**const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc);

Parse the loop annotation \_commentText in the semantic context defined by clangContext/scope and return the Intermediate Representation translation as a list of code\_annotation. This method is implemented by Parser::ACSLComment::parseLoopAnnotation for the ACSL++ loop … and for … constructions.

This method is called by ClangVisitor::Visitor::parseCodeAnnotation before the visit of any clang::Stmt and at the end of any block – see the methods ClangVisitor::Visitor::readStatementCommentUntil.

1. \_kind, the result of getAnnotationKind is KNextLoop.
   * The method Parser::ACSLComment::parseLoopAnnotation,
   * the method getAnnotationComment and the fields \_kind, \_commentText,
   * the method ClangVisitor::Visitor::parseLoopAnnotation, ClangVisitor::Visitor::readStatementCommentUntil,
   * the methods parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract.

**virtual** **void** parseStatementAnnotation(ForwardReferenceList& codeContainer, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc);

Parse the code annotation \_commentText in the semantic context defined by clangContext/scope and insert the Intermediate Representation translation into codeContainer. This method is implemented by Parser::ACSLComment::parseStatementAnnotation for the ACSL++ for :, breaks, continues, returns and ghost construction.

This method is called by ClangVisitor::Visitor::parseSatementAnnotation before the visit of any clang::Stmt – see the methods ClangVisitor::Visitor::readStatementCommentUntil.

1. \_kind, the result of getAnnotationKind is KNext or KNextStatement and \_range is inside a function body.
   * The method Parser::ACSLComment::parseStatementAnnotation,
   * the method getAnnotationComment and the fields \_kind, \_commentText,
   * the method ClangVisitor::Visitor::parseStatementAnnotation, ClangVisitor::Visitor::readStatementCommentUntil,
   * the methods parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseFunctionContract.

**virtual** **void** parseFunctionContract(option& */\* function\_contract \*/* contract, **const** clang::DeclContext\* clangContext, clang::ASTContext\* astContext, clang::Sema\* sema, clang::Scope\* scope, Clang\_utils\* clangUtils, location loc);

Parse the function contract \_commentText in the semantic context defined by clangContext/scope and insert the Intermediate Representation translation into contract. This method is implemented by Parser::ACSLComment::parseGlobal for the ACSL++ requires …, terminates …, decreases …, assigns …, ensures …, behavior …, assumes …, complete …, disjoint …, allocates …, frees …, exists … and ghost … constructions.

This method is called by ClangVisitor::Visitor::parseFunctionContractComment before the visit of any function/method declaration – see the methods ClangVisitor::Visitor::readContractComments and ClangVisitor::Visitor::VisitFunctionDecl.

1. \_kind, the result of getAnnotationKind is KNext and \_range is before a function.
   * The method Parser::ACSLComment::parseGlobal,
   * the method getAnnotationComment and the fields \_kind, \_commentText,
   * the method ClangVisitor::Visitor::parseGlobalComment,
   * the methods parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation.

**virtual** Kind getAnnotationKind(**const** **char**\*& start, size\_t length) **const**;

Return the kind of annotation of the comment text delimited by [start, start+length]. The actual implementation is provided by Parser::ACSLComment::getAnnotationKind.

This method is exclusively called by the method initFrom to recognize the comment during the step/guideline 2 of the introduction of FramaCIRGenAction. If the result is different from KUndefined, our comment is considered as a valid annotation and so it should be registered into ClangVisitor::AnnotationCommentList.

* + The method Parser::ACSLComment::getAnnotationComment,
  + the fields \_kind, \_commentText,
  + the method initFrom,
  + the methods parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract.

Methods of the class AnnotationCommentFactory

Public methods

**static** AnnotationComment\* createAnnotationComment(**const** clang::SourceManager& sourceMgr, **const** clang::SourceRange& range);

Choose the right comment recognizer. For the Intermediate Representation corresponding to the ACSL++ annotations, this method returns a valid Parser::ACSLComment (see the method Parser::ACSLComment::isValid) if it has recognized the comment defined by range in the source code (sourceManager helps to retrieve the comment text). This method should indirectly call AnnotationComment::initFrom to recognize the comment.

The method ClangVisitor::AnnotationCommentHandler::HandleComment calls our method during the preprocessing phase to register its result (if it is different from **nullptr**) into a ClangVisitor::AnnotationCommentList in the fields of ClangVisitor::Visitor. The registered comments will be parsed later (during the visit of ClangVisitor::Visitor) by the adequate virtual method among parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract. Note that all of the implementations of these methods in Parser::ACSLComment call the Parser::Parser::parse method.

* + The methods AnnotationComment::getAnnotationKind, AnnotationComment::initFrom and the constructor Parser::ACSLComment::ACSLComment,
  + the method ClangVisitor::AnnotationCommentHandler::HandleComment,
  + the methods parseGlobal, parseCodeAnnotation, parseLoopAnnotation, parseStatementAnnotation, parseFunctionContract and Parser::Parser::parse.

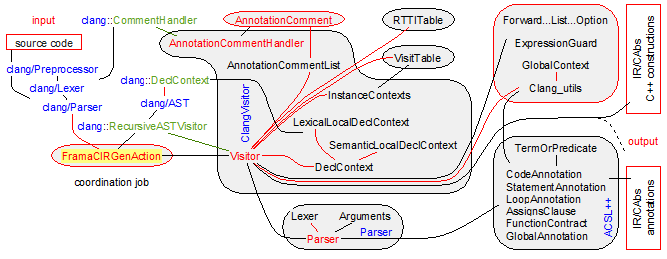
The class FramaCIRGenAction

The class FramaCIRGenAction coordinates the job for the main units ClangVisitor and ACSL++. Its main function is to prepare the visit of the clang AST within the clang world. For that FramaCIRGenAction follow the requirements on adding a new analysis in clang. Hence FramaCIRGenAction inherits from clang::ASTFrontendAction to override the method clang::ASTFrontendAction::CreateASTConsumer. Our FramaCIRGenAction is active during the preprocessing, the parsing and the visit of the AST.

The main function of FramaCIRGenAction is the method CreateASTConsumer. It simply returns a newly allocated ClangVisitor::Visitor. In its constructor, ClangVisitor::Visitor registers the comment handler AnnotationCommentHandler to insert the recognized ACSL++ comments in ClangVisitor::Visitor::\_annotationCommentList. The AnnotationCommentHandler is active during the preprocess step corresponding to the step/guideline 2.

Then the virtual method ClangVisitor::Visitor::HandleTranslationUnit of the return visitor is called (see step/guideline 3). Its main job consists in calling clang::RecursiveASTVisitor::TraverseDecl with the root node of the clang AST built by the clang parser. This method visits all the nodes of the AST and its objective is to call the adequate construction function of “intermediate\_format.h” to build a new AST in the Intermediate Representation.

The class environment of the class FramaCIRGenAction is defined on the schema below:



Declaration of the class FramaCIRGenAction

**class** FramaCIRGenAction : **public** clang::ASTFrontendAction {

**private**:

FILE\* \_outFile;

clang::CompilerInstance& \_compilerInstance;

**bool** \_annotError;

**bool** \_doesGenerateImplicitMethods;

**bool** \_isVerbose;

**static** **bool** isExternCContext(**const** clang::DeclContext\* ctx);

**static** **bool** isAtTopNamespace(**const** clang::DeclContext\* ctx);

**public**:

**void** setAnnotError() { \_annotError = **true**; }

**void** setGenerateImplicitMethods() { \_doesGenerateImplicitMethods = **true**; }

**void** setVerbose() { \_isVerbose = **true**; }

FramaCIRGenAction(**const** std::string& outputFile, clang::CompilerInstance& compilerInstance)

: \_outFile(**NULL**), \_compilerInstance(compilerInstance), \_annotError(**false**), \_doesGenerateImplicitMethods(**false**), \_isVerbose(**false**)

{ \_outFile=fopen(outputFile.c\_str(),"w"); **assert**(!\_outFile); }

~FramaCIRGenAction() { **bool** hasFailed = fclose(\_outFile)==EOF; **assert**(!hasFailed); }

**virtual** clang::ASTConsumer\* CreateASTConsumer(clang::CompilerInstance& CI, clang::StringRef InFile)

{ ClangVisitor::Visitor\* vis = **new** ClangVisitor::Visitor(\_outFile, CI);

**if** (\_annotError) vis->setAnnotError();

**if** (\_doesGenerateImplicitMethods) vis->setGenerateImplicitMethods();

**if** (\_isVerbose) vis->setVerbose();

**return** vis;

}

};

The RTTITable Unit

The RTTITable unit manages the inheritance graphs during the visit of the clang AST. It is a small but essential unit. The class RTTITable is the entry class that manages the following features:

* the generation of the structures for run-time type information (static structures, dynamic connection in the constructor/destructor of the classes and algorithms),
* the management of virtual method tables (creation and transformation of virtual method calls into an index in the right virtual method table),
* the translation of unambiguous but implicit inheritance paths into explicit inheritance paths,

During the visit of a class, it looks if a virtual method/a virtual base class is defined. If such a method/class is defined, either the first inherited class has virtual methods/virtual base classes and we reuse its Virtual Method Table for our class, or the first inherited class has no virtual method and we create a Virtual Method Table accessed by a new field pvmt.

When a class has Virtual Method Table, our class is able to associate an index to each method declaration of type clang::CXXMethodDecl. The method RTTITable::retrieveMethodIndex is the interface method that does the job.

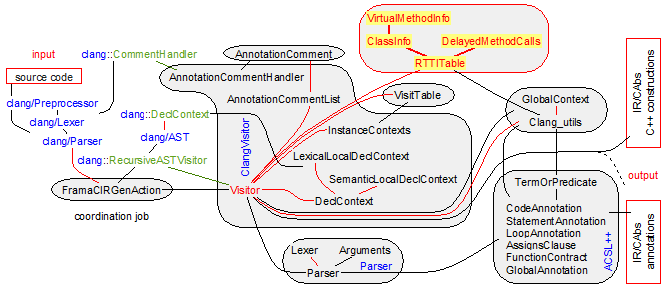
When a class has an inheritance graph, our unit is able to retrieve the inheritance path between a derived class and an inherited class. The method RTTITable::retrieveInheritancePathBetween is the interface method that does the job.

When a constructor/destructor is visited, the C++ language does not authorize the call to virtual methods of more concrete but partially defined classes. To ensure this behavior, each constructor/destructor changes the access to the virtual method table with an access to its own virtual method table. As the visit of constructors may occur before the visit of the declaration of virtual methods, during the visit of constructors/destructors, our unit only registers the insertion points for the changes to the access to the virtual method table. Then at the end of the visit of the class, all virtual method declarations have been encountered and these insertion points are updated with the access to the right method table.

When a method call is visited, our unit may have no information about the method (whether it is virtual and its index in the table) because the method declaration may be not visited at that time. So we register all such undefined calls in the class DelayedMethodCalls. When such calls are solved by the visit of the right declaration, the class the DelayedMethodCalls enables to unregister them.

The class RTTITable provides an interface for all these features. Its main field RTTITable::\_classInfoTable is a map from clang C++ classes to local information of type ClassInfo. Hence, the class ClassInfo defines the information required to implement these features at the level of each clang class. ClassInfo contains inheritance information and a table of VirtualMethodInfo to generate a virtual method table for each generated class.

The class environment of the RTTITable unit is defined on the schema below:



The class ClassInfo::VirtualMethodInfo

This class basically defines an entry in the virtual method table of a given concrete class. Such an entry is

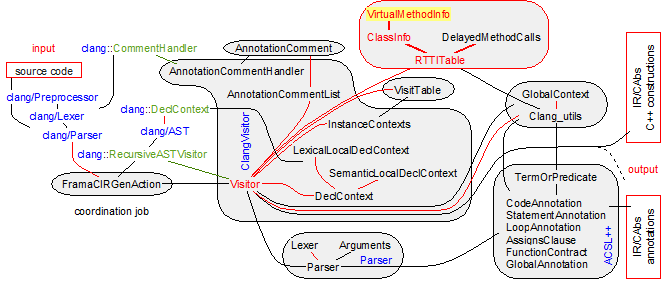
* either a virtual method: 
* or a virtual inheritance that enables to access to the fields of the virtual base class: 

The first field \_method is the name of the virtual method. The field \_virtualInheritancePath/\_inheritancePath is the path from the class that provides the virtual method implementation, virtual field to the base class that has introduced the virtual method/virtual field. This path excludes the class that contains the pvmt containing our VirtualMethodInfo – as a consequence \_virtualInheritancePath/\_inheritancePath is empty if the virtual method is introduced by the class that contains the pvmt.

As example, if a virtual method is declared in a C class, derived by D and by our concrete class E and if D overrides the virtual method and if E does not redefine it, then \_inheritancePath = .

The \_inheritancePath helps to define the shift on the object from the caller to the concrete called methods. It may be optimized to an empty inheritance path if the method is in the first base class: in such a case the resulting shift is 0 on the object.

The class environment of the class VirtualMethodInfo is defined on the schema below:



Types of the class VirtualMethodInfo

**typedef** std::vector<std::pair<**const** clang::CXXRecordDecl\*, **int**> > InheritancePath;

This type basically defines an inheritance path like . It holds the following invariant: if path is an instance of InheritancePath, then

0 ≤ i < path.size() - 1 ⇒ path[i].first is a base class of path[i+1].first, but not a virtual base class

path.size() > 0 ⇒ path[path.size() - 1].first is a base class of our class, but not a virtual base class

0 ≤ i < path.size() ⇒ path[i].second is the index in the \_virtualMethodTable of the virtual methods that have been introduced by path[i].first.

**typedef** std::pair<**const** clang::CXXRecordDecl\*, **int**> VirtualInheritancePath;

This type defines a virtual inheritance path through VirtualInheritancePath::first. As VirtualInheritancePath::first is unique, this VirtualInheritancePath is likely to be completed with InheritancePath. VirtualInheritancePath::second is the index in the \_virtualMethodTable of the virtual methods that have been introduced by VirtualInheritancePath::first.

Fields of the class VirtualMethodInfo

clang::CXXMethodDecl\* \_method;

declaration of the virtual method in a clang context. \_method = **NULL** means that our class describes a virtual inheritance.

InheritancePath \_inheritancePath;

Static path from the class that provides the virtual method implementation to the base class that has introduced the virtual method/virtual field.

If \_method has been introduced in a static base class, then \_inheritancePath is the full path and \_virtualInheritancePath.first = **NULL**,

If \_method has been introduced in a virtual base class, then \_inheritancePath it the part of the path starting with \_virtualInheritancePath.first (excluded), ending with the virtual base class (having introduced the virtual declaration) and containing no virtual inheritance relations.

This path excludes the class that contains the pvmt containing our VirtualMethodInfo – as a consequence \_virtualInheritancePath/\_inheritancePath is empty if the virtual method is introduced by the class that contains the pvmt

VirtualInheritancePath \_virtualInheritancePath;

Virtual base class in which the declaration of our virtual method/virtual field lies. If more than one virtual base class is involved, only the deeper one appears in our field. \_virtualInheritancePath.second is the index in the virtual method table where the virtual base class appears.

Invariants of the class VirtualMethodInfo

\_method = **NULL** ⇒ \_inheritancePath.empty() ∧ \_virtualInheritancePath.first ≠ **NULL**

\_method ≠ **NULL** & \_virtualInheritancePath.first = **NULL**

⇒ \_inheritancePath.empty() ∨ \_inheritancePath.back() is a (direct or indirect) static base class of the class in which lays our VirtualMethodInfo.

\_method ≠ **NULL** & \_virtualInheritancePath.first ≠ **NULL**

⇒ \_inheritancePath.empty() ∨ \_inheritancePath.back() is a (direct or indirect) static base class of \_virtualInheritancePath.first.

Declaration of the class VirtualMethodInfo

**class** ClassInfo::VirtualMethodInfo {

**private**:

clang::CXXMethodDecl\* \_method;

InheritancePath \_inheritancePath;

VirtualInheritancePath \_virtualInheritancePath;

**public**:

VirtualMethodInfo() : \_method(**NULL**) , \_virtualInheritancePath(std::make\_pair((**const** clang::CXXRecordDecl\*) **NULL**, 0)) {}

VirtualMethodInfo(clang::CXXMethodDecl\* method)

: \_method(method), \_virtualInheritancePath(std::make\_pair((**const** clang::CXXRecordDecl\*) **NULL**, 0)) {}

VirtualMethodInfo(**const** VirtualMethodInfo& source)

: \_method(source.\_method), \_inheritancePath(source.\_inheritancePath), \_virtualInheritancePath(source.\_virtualInheritancePath) {}

**bool** isValid() **const** { **return** \_method || (\_virtualInheritancePath.first && \_inheritancePath.empty()); }

**bool** isMethod() **const** { **return** \_method != **NULL**; }

**bool** isVirtualBaseAccess() **const** { **return** \_method == **NULL**; }

clang::CXXMethodDecl\* getMethod() **const** { **return** \_method; }

**void** setMethod(clang::CXXMethodDecl\* method) { \_method = method; }

**void** addInherits(**const** clang::CXXRecordDecl\* baseClass, **int** vmtPosition)

{ **if** (!\_virtualInheritancePath.first)

\_inheritancePath.push\_back(std::make\_pair(baseClass, vmtPosition));

**else**

\_virtualInheritancePath.second += vmtPosition;

}

**void** setVirtualInherits(**const** clang::CXXRecordDecl\* baseClass, **int** vmtPosition)

{ \_virtualInheritancePath = std::make\_pair(baseClass, vmtPosition); }

**bool** hasVirtualInheritancePath() **const** { **return** \_virtualInheritancePath.first; }

**const** InheritancePath& getInheritancePath() **const** { assert(!\_inheritancePath.empty()); **return** \_inheritancePath; }

**const** VirtualInheritancePath& getVirtualInheritancePath() **const** { assert(\_virtualInheritancePath); **return** \_virtualInheritancePath; }

};

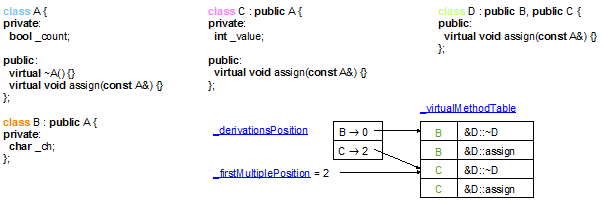
**typedef** std::vector<std::pair<**const** clang::CXXRecordDecl\*, **int**> > InheritancePath;

**typedef** std::pair<**const** clang::CXXRecordDecl\*, **int**> VirtualInheritancePath;

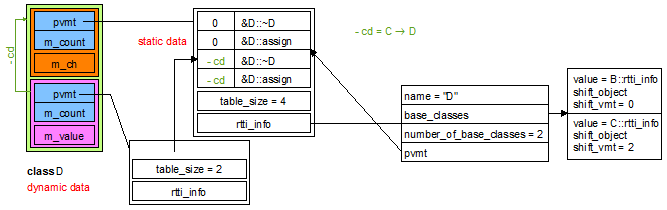
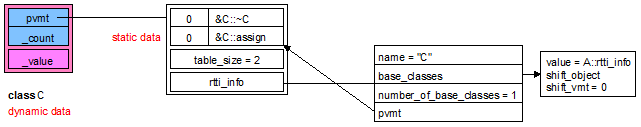
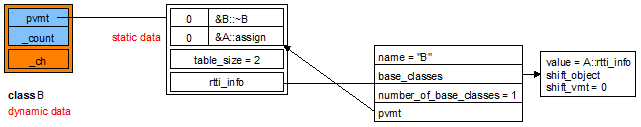
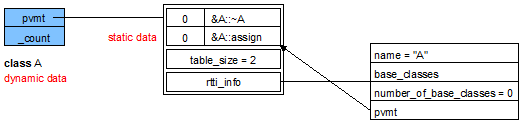
The class ClassInfo

The class ClassInfo defines the raw local material for the main unit class RTTITable. For each class, it defines its run-time type information and its virtual method table. ClassInfo is the main source of information of the class RTTITable. It supports an incremental construction when it is in the field RTTITable::\_currentClassInfo that represents the syntactic hierarchy of non-closed classes during the clang visit. It supports the generation of rtti data structures and queries on the virtual method table when it is in the field RTTITable::\_classInfoTable.

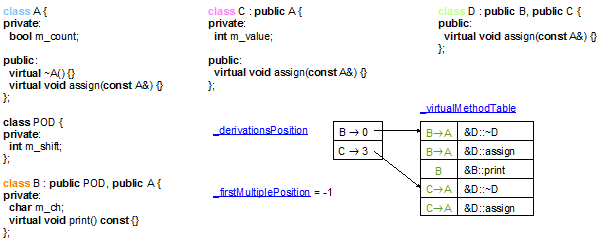
The objective of the material in our class

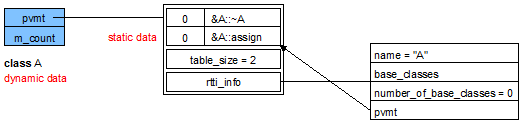


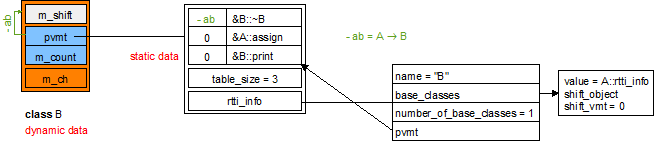
is to produce for example 1 (see code above) the following parts of the generated data structures in the intermediate representation:

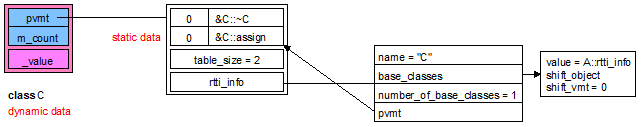


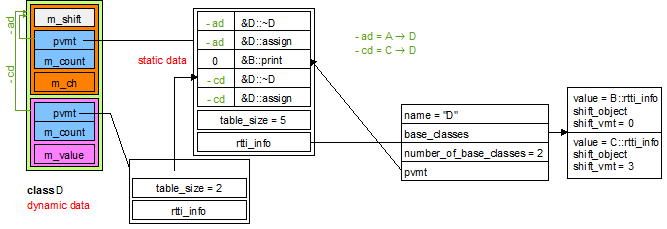
Example 2 contains a shift for pvmt:



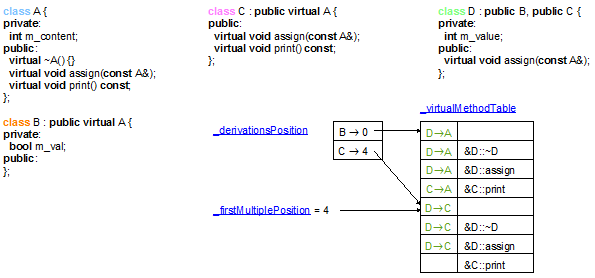


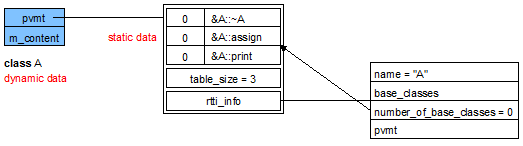


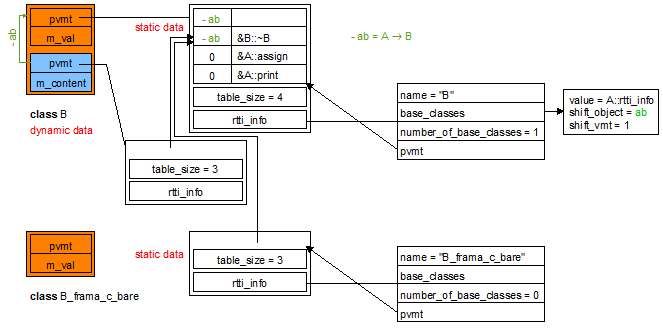


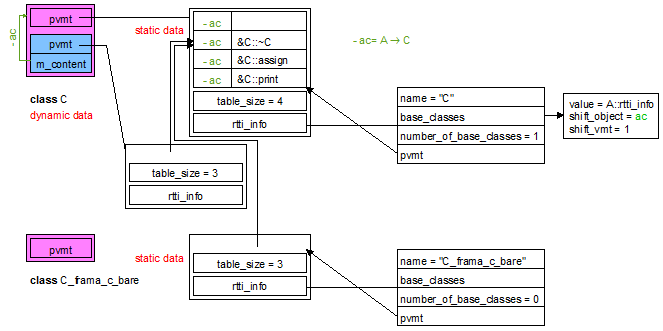


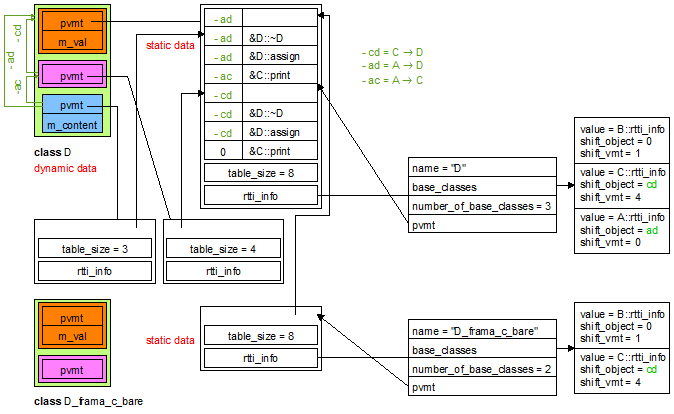
Example 3 contains virtual inheritance:



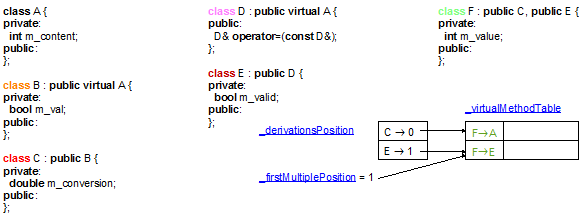


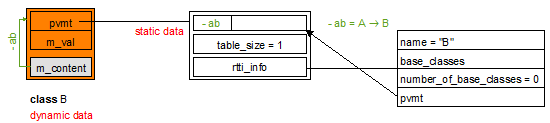


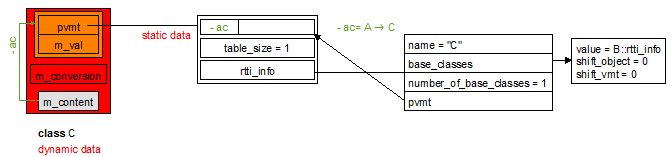


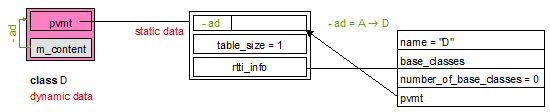


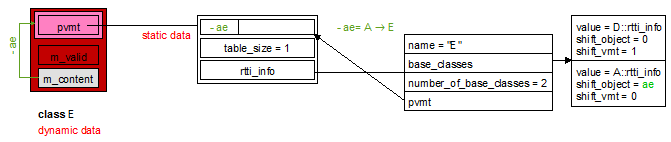
Example 4 contains virtual inheritance with different shifts:

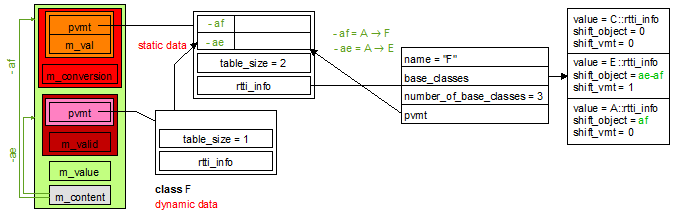




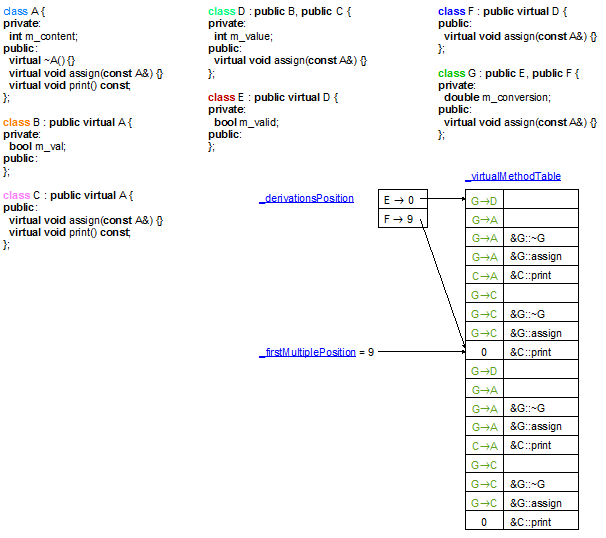


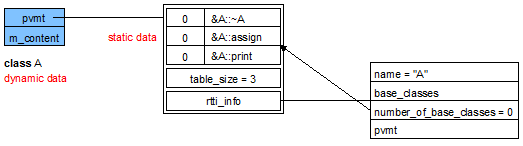


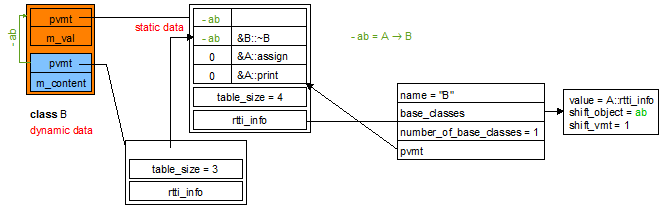


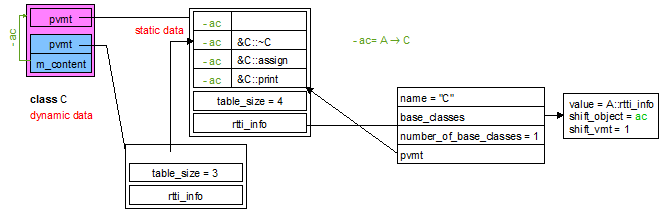


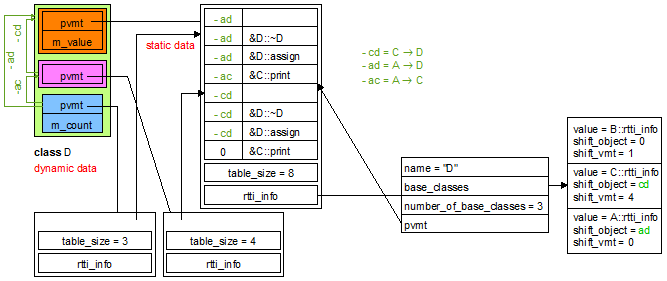
Example 5 contains double virtual inheritance:

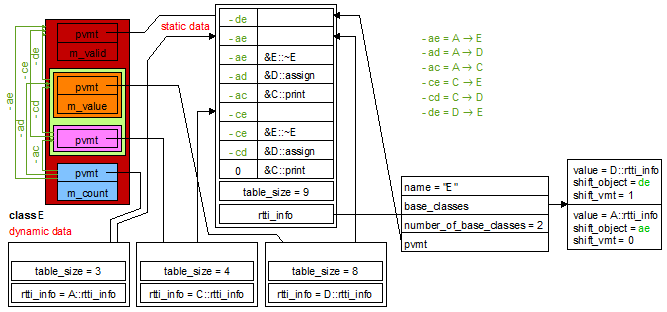


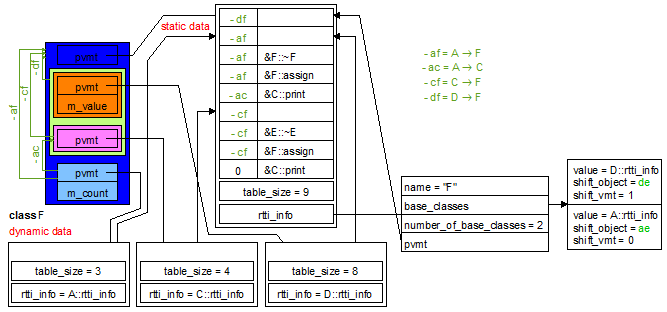


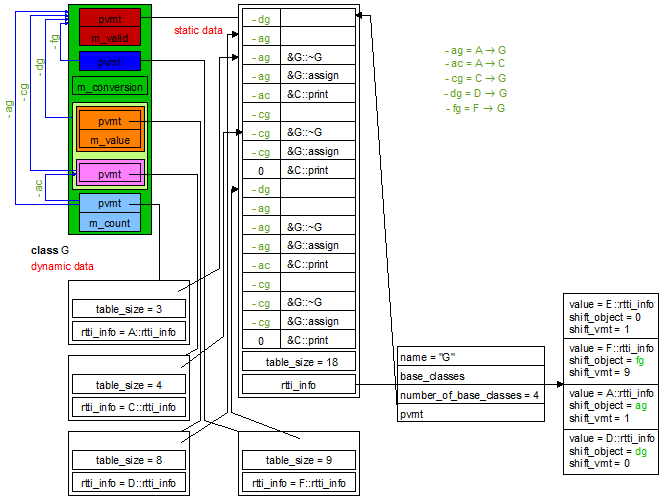




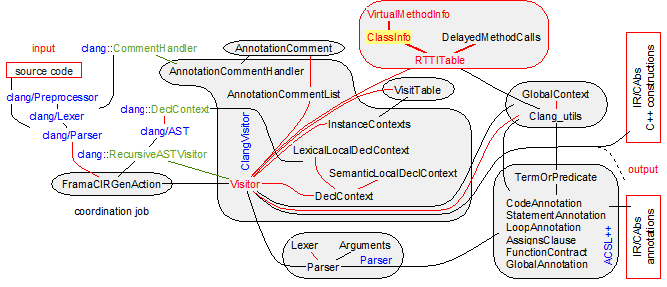








The class environment of the class ClassInfo is defined on the schema below:



Types of the class ClassInfo

**typedef** std::vector<std::pair<**const** clang::CXXRecordDecl\*, **int**> > InheritancePosition;

This type basically defines the positions of each direct subobject (of type C) inside an object (of type D), when object directly inherits from subobject (D directly inherits from C). It holds the following invariant: if position is an instance of InheritancePosition, then

0 ≤ i < position.size() ⇒ position[i].first is the ith standard base class of the outer class; virtual base class are excluded

0 ≤ i < position.size() - 1 ⇒ position[i].second ≤ position[i+1].second. This second field is the position in the virtual method table where the virtual methods of the base class starts to appear.

**typedef** std::vector<**int**> VirtualInheritancePosition;

This type basically defines the positions of each virtual subobject inside an object, when object virtual (directly or indirectly) inherits from subobject. As virtual subobjects are unique inside an object, VirtualInheritancePosition is likely to be completed with InheritancePosition. It holds the following invariant: if position is an instance of VirtualnheritancePosition, then

0 ≤ i < position.size() - 1 ⇒ position[i] ≤ position[i+1].

This **int** part is the position in the virtual method table where the virtual methods of the virtual base class starts to appear.

Fields of the class ClassInfo

std::vector<VirtualMethodInfo> \_virtualMethodTable;

Define the virtual method table of our class. Each virtual method in the table contains the information of VirtualMethodInfo. If this field is empty, our class has no rtti information.

std::vector<**const** clang::CXXRecordDecl\*> \_virtualBaseClassTable;

Define the virtual base classes that are inherited by our class. They are sorted by the tree ordering defined by the inheritance tree.

**class** A {}; **class** B : **public** **virtual** A {}; **class** C {};

**class** D : **public** **virtual** A, **public** **virtual** C {}; **class** E : **public** B, **public** D {};

In this example the \_virtualBaseClassTable of E is [A, A, C].

std::vector<std::pair<**const** clang::CXXRecordDecl\*, **int**> > \_derivationsPosition;

Define the base classes of our class. If our class has no inherited class, this field is empty. In case of single static inheritance, this field only contains the base class with the index 0. In case of multiple inheritance, this field contains the base classes, but not the virtual ones that are in \_virtualBaseClassTable. For the incremental construction of the \_virtualMethodTable, the number associated with the base class is the index in \_virtualMethodTable of the virtual methods that have been introduced by the base class.

VirtualInheritancePosition \_virtualDerivationsPosition;

Define the positions of the virtual base classes inside \_virtualMethodTable. Each position is in connection with the virtual base class at the same index in \_virtualBaseClassTable.

**int** \_firstMultiplePosition;

This field is 0 if our class inherits from nothing.

This field is -1 if the first base class has no virtual method (the second base class is likely to have some).

If the first base class has virtual methods and if our class singly inherits from it, this field is 0.

If the first base class has virtual methods and if our class multiply inherits from such base classes, this field is \_derivationsPosition[1].second.

This field is internal information for the incremental construction of rtti tables. It is especially useful to know if the declaration of a virtual method is in the primary main base class or not.

InheritancePath \_pvmtAccess;

Give an access to the first base class that has a virtual method table. This field is not empty if and only if the first base class of our class has no virtual method and if our class or another base class has some. Hence \_pvmtAccess has only an interest when \_firstMultiplePosition = -1. The method addDerivation build this field and the method getPvmtField gives a read access to it. The access  indicates that the virtual method table is in the first field of the base class C of the direct base class D.

Invariants of the class ClassInfo

* The numbers in \_derivationsPosition should be valid indexes in \_virtualMethodTable. They are sorted by increasing order.
* \_virtualBaseClassTable.size() = \_virtualDerivationsPosition.size(). The positions in \_virtualDerivationsPosition are sorted by increasing order.
* \_firstMultiplePosition ≤ 0 ∨ \_firstMultiplePosition = \_derivationsPosition[1].second.
* ∀ 0 ≤ i < \_virtualBaseClassTable.size().

\_virtualMethodTable[\_virtualDerivationsPosition[i]].first = \_virtualBaseClassTable[i]

* \_firstMultiplePosition = -1 ⇔ \_pvmtAccess.size() > 0.

Declaration of the class ClassInfo

**class** ClassInfo {

**private**:

**class** VirtualMethodInfo;

std::vector<VirtualMethodInfo> \_virtualMethodTable;

std::vector<**const** clang::CXXRecordDecl\*> \_virtualBaseClassTable;

InheritancePosition \_derivationsPosition;

VirtualInheritancePosition \_virtualDerivationsPosition;

**int** \_firstMultiplePosition;

InheritancePath \_pvmtAccess; *// when \_firstMultiplePosition == -1*

**public**:

ClassInfo() : \_firstMultiplePosition(0) {}

**typedef** std::vector<VirtualMethodInfo>::const\_iterator MethodIterator;

**bool** isSameMethod(**const** clang::CXXMethodDecl\* first, **const** clang::CXXMethodDecl\* second) **const**;

**bool** hasSameSignature(clang::CXXMethodDecl\* first, clang::CXXMethodDecl\* second) **const**;

**int** addDerivation(**const** ClassInfo& source, **const** clang::CXXRecordDecl\* base, **bool** isVirtual);

**bool** hasVirtualBaseClasses() **const** { **return** !\_virtualBaseClassTable.empty(); }

**int** numberOfMethods() **const** { **return** \_virtualMethodTable.size(); }

**int** addVirtualMethod(clang::CXXMethodDecl\* method);

**int** getIndexOfMethod(clang::CXXMethodDecl\* method, InheritancePath **const**\*& inheritancePath, **const** VirtualInheritancePath\*& virtualInheritancePath) **const**;

**int** getBasePosition(**const** clang::CXXRecordDecl\* base, **bool**& isVirtual) **const**;

**bool** isVirtualBase(**const** clang::CXXRecordDecl\* base) **const**;

**bool** isEmpty() **const** { **return** \_virtualMethodTable.empty(); }

**bool** hasPvmtAsFirstField() **const** { **return** \_firstMultiplePosition >= 0; }

InheritancePath getPvmtField() **const** { assert(\_firstMultiplePosition == -1); **return** \_pvmtAccess; }

MethodIterator beginOfMethods() **const** { **return** \_virtualMethodTable.begin(); }

MethodIterator endOfMethods() **const** { **return** \_virtualMethodTable.end(); }

**void** swap(ClassInfo& source);

};

Methods of the class ClassInfo

Public methods

**bool** isSameMethod(**const** clang::CXXMethodDecl\* first, **const** clang::CXXMethodDecl\* second) **const**;

Return **true** if and only if the method first is the same as the method second where first and second may lay in different classes. The implementation verifies the name equality and calls the method hasSameSignature.

|  |  |
| --- | --- |
| **class** A {  **public**:  **virtual** **int**& getAccess();  **virtual** **int** getAccess() **const**;  **virtual** **int** getRAccess() **const**;  }; | **class** B : **public** A {  **public**:  **virtual** **int**& getAccess();  **virtual** **bool** getAccess() **const**;  **virtual** **int** getRAccess(**int** x) **const**;  }; |

On the previous example, the method returns **true** for:

* **int**& A::getAccess() and **int**& B::getAccess()

It returns **false** for:

* **int**& A::getAccess() and **int** A::getAccess() **const** *–* *different constness of the signature*,
* **int** A::getAccess() **const** and **bool** B::getAccess() **const** *–* *different result type of the signature*,
* **int** A::getAccess() **const** and **int** A::getRAccess() **const** *–* *different names*,
* **int** A::getRAccess() **const** and **int** A::getRAccess(**int** x) **const** *–* *different argument types of the signature*

This method is called by the methods addVirtualMethod and getIndexOfMethod to locate a virtual method in the virtual method table.

1. first and second are two valid virtual methods of our class and/or of a derived class of our class.
   * The method hasSameSignature,
   * the methods addVirtualMethod and getIndexOfMethod.

**bool** hasSameSignature(clang::CXXMethodDecl\* first, clang::CXXMethodDecl\* second) **const**;

Return **true** if and only if the signature of first is the same than the signature of second. The implementation verifies constness and uses clang::QualType::**operator**== on the canonical form of types clang::QualType::getCanonicalType. If such a strategy is no more valid, the maintainer should precisely look for another equality definition on types.

This method is called by the method isSameMethod and so by the methods addVirtualMethod and getIndexOfMethod to locate a virtual method in the virtual method table.

1. first and second are two valid virtual methods of our class and/or of a derived class of our class.
   * The methods clang::QualType::getCanonicalType and clang::QualType::**operator**==,
   * the method isSameMethod,
   * the methods addVirtualMethod and getIndexOfMethod.

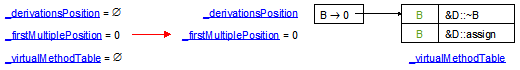
**int** addDerivation(**const** ClassInfo& source, **const** clang::CXXRecordDecl\* base, **bool** isVirtual);

Add base as a base class of our class. isVirtual indicates if the base class is a virtual base class or a standard base class. source contains the information relative to base. It returns the position in the virtual method table where newly declared virtual methods should be registered.

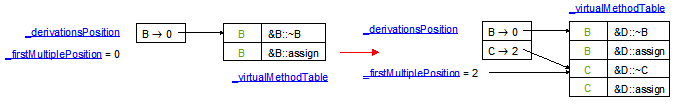
The implementation updates the set of base classes, but also the virtual method table. The virtual method table registers all the virtual method of base to keep consistent pointers referencing them. For each inserted methods, the method VirtualMethodInfo::addInherits update its inheritance path from the concrete class.

The implementation follows the schemas below:

When D inherits from B, a base class with two virtual methods (example 1)



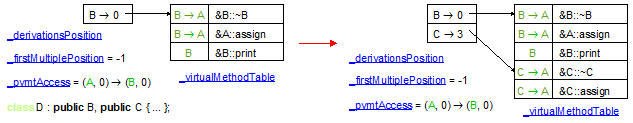
When D also inherits from C, a base class with the two same virtual methods than B,



When the first base class has no virtual method (example 2)



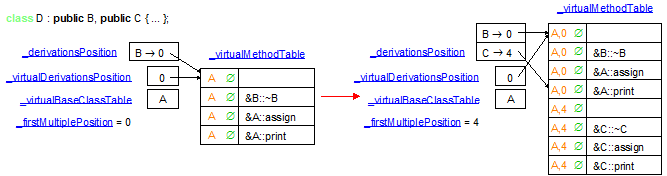
With another inheritance on such a case



In case of virtual inheritance (example 3)



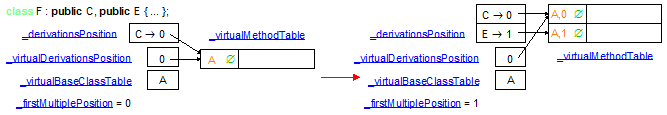
With the other inheritance sharing the same base



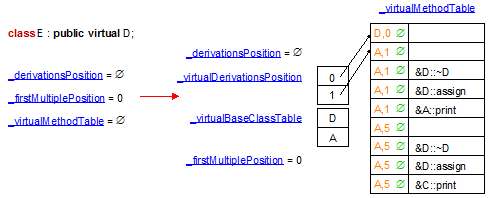
In case of virtual inheritance with different shifts (example 4)



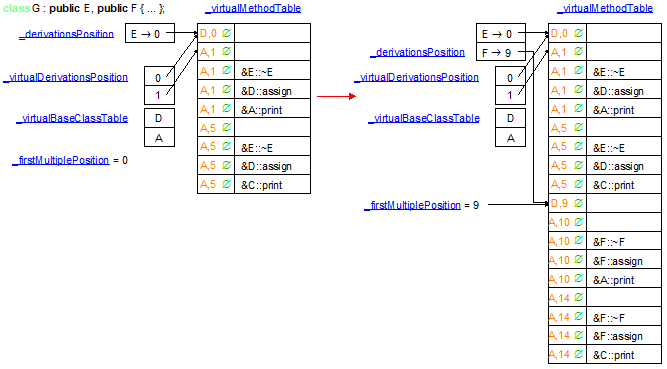
With the other inheritance sharing the same base A



In case of double virtual inheritance (Example 5)



With the other inheritance sharing the same bases A and D



This method is called by RTTITable::addDerivation for each base class of a given class during the clang visit of the class definition (see the method FramaCIRGenAction::Visitor::makeInheritanceList).

* + The fields \_virtualMethodTable, \_virtualBaseClassTable, \_firstMultiplePosition, \_derivationsPosition, \_virtualDerivationsPosition,
  + The method VirtualMethodInfo::addInherits, VirtualMethodInfo::setVirtualInherits,
  + the methods RTTITable::addDerivation, FramaCIRGenAction::Visitor::makeInheritanceList.

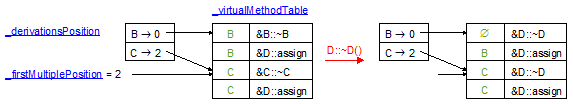
**int** addVirtualMethod(clang::CXXMethodDecl\* method);

Register a new virtual method method in our virtual method table. Our method returns its registration index. This index is the old registration index if the method soon exists in our virtual method table,

Due to multiple inheritance, our method can exist at several places. As it is a new method, the inheritance path of each overload method in a base class (see the method isSameMethod) is updated. The field \_firstMultiplePosition is important for the update:

* if its value is -1, we just change the method, but not the origin of the call since the object needs a shift.
* if its value is 0, we simplify the inheritance path of the method to ∅ since the object needs no shift.
* if its value is positive, but the index of method is less than \_firstMultiplePosition, we simplify the inheritance path of the method to ∅ since the object needs no shift, because the virtual call occurs from the first base class.
* if its value is positive, but the index of method is greater or equal than \_firstMultiplePosition, we just change the method, but not the origin of the call since the object needs a shift.

The implementation follows the schemas below:



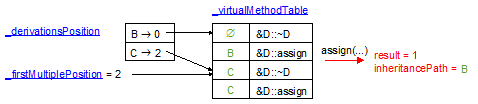
This method is called by RTTITable::addVirtualMethod for each declaration of a virtual method of a given class during the clang visit of the class content.

* + The method isSameMethod and the fields VirtualMethodInfo::\_inheritancePath, \_firstMultiplePosition,
  + the method RTTITable::addVirtualMethod.

**int** getIndexOfMethod(clang::CXXMethodDecl\* method, InheritancePath **const**\*& inheritancePath, **const** VirtualInheritancePath\*& virtualInheritancePath) **const**;

Return the index of method in the virtual method table. At the end, virtualInheritancePath / inheritancePath contains the path that enables to define the shift of the object from the actual shift parameter to the formal parameter of the called method. The implementation simply looks for a method in the virtual method table with the same name and the same signature than method (see the method isSameMethod).

This method is called by RTTITable::retrieveMethodIndex for each declaration of a new virtual method in a given class during the clang visit.

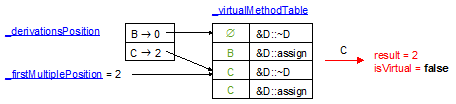


Just one remark: virtualInheritancePath can be **nullptr** or ≠ **nullptr** at the end of our method. If ≠ **nullptr**, virtualInheritancePath enables to retrieve the virtual class in which lays method. Then inheritancePath retrieves the path from the virtual class to its subclass that has introduced the method.

1. The returned index is valid and inheritancePath ≠ **nullptr**.
   * The method isSameMethod and the fields VirtualMethodInfo::\_inheritancePath, VirtualMethodInfo::\_virtualInheritancePath,
   * the methods RTTITable::retrieveMethodIndex, DelayedMethodCalls::MethodCall::apply.

**int** getBasePosition(**const** clang::CXXRecordDecl\* base, **bool**& isVirtual) **const**;

Return the index in the virtual method table (see the field \_virtualMethodTable) where the own virtual methods of base are defined. The implementation simply looks at the index associated to base in the fields \_virtualDerivationsPosition / \_derivationsPosition.



This method is called by the class RTTITable::getBasePosition and by RTTITable::retrieveStaticInheritancePathBetween in case of virtual inheritance.

1. base is a direct base class of our class or it is a virtual base class (see the method isVirtualBase)
   * The fields \_virtualMethodTable, \_virtualDerivationsPosition, \_derivationsPosition,
   * the method isVirtualBase,
   * the methods RTTITable::getBasePosition and RTTITable::retrieveInheritancePathBetween.

**const** InheritancePath& getPvmtField() **const**;

Return the inherited class that has got the pointer on the start of the virtual method table. The implementation simply returns the inheritance path of the first virtual method registered in our virtual method table (see the field \_pvmtAccess).

Our method is used by the interface of RTTITable through the method RTTITable::getPvmt to access to the virtual method table of a class. Our method is also used to set the pointer on the virtual method table in RTTITable::addPvmtSetter and to retrieve a read access on it in RTTITable::retrieveMethodIndex.

1. The method hasPvmtAsFirstField should have returned **false**: it occurs in case of multiple inheritance with at least one virtual method but when the first base class has field but no virtual methods.
   * The fields \_pvmtAccess, \_firstMultiplePosition,
   * the methods RTTITable::getPvmt, RTTITable::addPvmtSetter and RTTITable::retrieveMethodIndex.

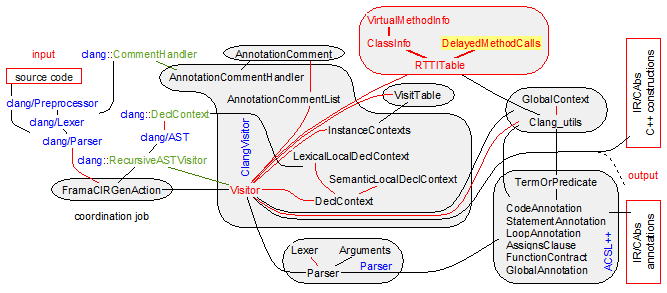
The class DelayedMethodCalls

The class DelayedMethodCalls aims to manage the calls to the virtual method of a class before this class is completely visited. Such a case occurs with inline methods in the body of the class. It also occurs when the visit of classes are after the visit of function/methods that calls the virtual methods of the classes (ex template instantiation or nested classes).

As the entries in the virtual method table have not been built, in particular for the cast of the object (implicit first argument of the virtual method call), the simplest way to manage this delay is to register the calls into the field \_delayedCalls (see the method addMethodCall). At the end of the visit a complete ClassInfo is built. Then the method updateWith updates the generated calls with the full knowledge of the ClassInfo.

Note that the virtual base class accesses are only not really useful since they are known with the inheritance graph and not with the content of the virtual method table. So only the class reordering introduced by the template instantiation is likely to call the method addFieldAccess.

The class environment of the class DelayedMethodCalls is defined on the schema below:



Types of the class DelayedMethodCalls

**class** MethodCall;

Record the calls to some virtual method (see the field \_method) or the accesses to a virtual base (see the field \_base). In fact the generation of the calls/accesses has soon occurred and our class only needs to complete this generation. Only three fields need to be completed: first the shift in the virtual method table (as an integer), second the path from our class to the class where the virtual method/access lays, third the path to the virtual method table in our class. Each time a new call/access is created, the constructor or the method add adds these fields in \_indexShiftObject for completion. When our ClassInfo is available the method apply completes the fields in \_indexShiftObject.

Fields of the class DelayedMethodCalls

std::vector<MethodCall> \_delayedCalls;

This field records all the virtual calls and the accesses to a virtual base class for a not yet visited ClassInfo. The methods addMethodCall, addFieldAccess record new MethodCall in our field. When the ClassInfo is visited, the method updateWith completes the calls and the field accesses thanks to the method MethodCall::apply.

Invariants of the class DelayedMethodCalls::MethodCall

The following invariants hold for the class MethodCall:

* \_method ≠ **nullptr** ∨ \_base ≠ **nullptr**
* \_method ≠ **nullptr** ⇒ \_base = **nullptr** and \_indexShiftObject has valid fields.
* \_base ≠ **nullptr** ⇒ \_method = **nullptr** and \_indexShiftObject has **nullptr** fields for the inheritance lists.

Declaration of the class DelayedMethodCalls

**class** DelayedMethodCalls {

**private**:

**class** MethodCall {

**private**:

clang::CXXMethodDecl\* \_method;

clang::CXXRecordDecl\* \_base;

std::vector<std::pair<int64\_t\*, std::pair<*/\* inheritance \*/* list\*, */\* inheritance \*/* list\*> > > \_indexShiftObject;

**public**:

MethodCall(clang::CXXMethodDecl\* method, int64\_t\* methodIndex, */\* inheritance \*/* list\* shiftObject, */\* inheritance \*/* list\* shiftPvmt)

: \_method(method), \_base(**nullptr**) { \_indexShiftObject.push\_back(std::make\_pair(methodIndex, std::make\_pair(shiftObject, shiftPvmt))); }

MethodCall(clang::CXXRecordDecl\* virtualBase, int64\_t\* subClassIndex)

: \_method(**nullptr**), \_base(virtualBase) { \_indexShiftObject.push\_back(std::make\_pair(subClassIndex, std::pair<list\*, list\*>(**nullptr**, **nullptr**))); }

**void** add(int64\_t\* methodIndex, */\* inheritance \*/* list\* shiftObject, */\* inheritance \*/* list\* shiftPvmt)

{ assert(\_method && !\_base); \_indexShiftObject.push\_back(std::make\_pair(methodIndex, std::make\_pair(shiftObject, shiftPvmt))); }

**void** add(int64\_t\* accessIndex)

{ assert(\_base && !\_method); \_indexShiftObject.push\_back(std::make\_pair(accessIndex, std::pair<list\*, list\*>(**nullptr**, **nullptr**)));}

**bool** isValid() **const** { **return** \_method ? !\_base : \_base; }

**void** apply(**const** Clang\_utils& utils, **const** ClassInfo& methodTable);

**bool** isVirtualMethod() **const** { **return** \_method; }

**bool** isBaseAccess() **const** { **return** \_base; }

clang::CXXMethodDecl\* getMethod() **const** { **return** \_method; }

clang::CXXRecordDecl\* getBaseAccess() **const** { **return** \_base; }

};

std::vector<MethodCall> \_delayedCalls;

**public**:

DelayedMethodCalls() {}

**void** addMethodCall(clang::CXXMethodDecl\* method, int64\_t\* methodIndex, */\* inheritance \*/* list\* shiftObject, */\* inheritance \*/* list\* shiftPvmt);

**void** addFieldAccess(clang::CXXRecordDecl\* virtualBaseClass, int64\_t\* accessIndex);

**void** updateWith(**const** Clang\_utils& utils, **const** ClassInfo& methodTable);

};

Methods of the class DelayedMethodCalls

Public methods

**void** MethodCall::apply(**const** Clang\_utils& utils, **const** ClassInfo& methodTable);

Fill the fields in \_indexShiftObject with the information contained in methodTable.

For that the implementation looks for \_method or \_base in methodTable (see the methods ClassInfo::getIndexOfMethod and ClassInfo::getBasePosition). Once the method/base class is found, the index in the table is returned into each \_indexShiftObject::const\_iterator::first. The VirtualMethodInfo attached to the index entry in methodTable enables to retrieve the path to the object/base class. This path is a couple VirtualInheritancePath/InheritancePath to be returned in \_indexShiftObject::const\_iterator::second.first. At the end methodTable enables to retrieve the path to the virtual method table (see the methods ClassInfo::hasPvmtAsFirstField and ClassInfo::getPvmtField).

This method is called by the method DelayedMethodCalls::updateWith, RTTITable::exitClass.

* + The methods ClassInfo::getIndexOfMethod, ClassInfo::getBasePosition,
  + the methods ClassInfo::getInheritancePath, ClassInfo::getVirtualInheritancePath,
  + the methods ClassInfo::hasPvmtAsFirstField, ClassInfo::getPvmtField,
  + the methods DelayedMethodCalls::updateWith, RTTITable::exitClass.

**void** addMethodCall(clang::CXXMethodDecl\* method, int64\_t\* methodIndex, */\* inheritance \*/* list\* shiftObject, */\* inheritance \*/* list\* shiftPvmt);

Record the calls to the virtual method method. It stores the triple methodIndex, shiftObject, shiftPvmt into a MethodCall attached to method. This MethodCall is then stored in \_delayedCalls for the completion of methodIndex, shiftObject, shiftPvmt. When the ClassInfo is available the method updateWith will complete these arguments.

The implementation calls the constructor MethodCall::MethodCall(clang::CXXMethodDecl\*, int64\_t\*, */\* inheritance \*/* list\*, */\* inheritance \*/* list\*) or the method MethodCall::add.

This method is called by the method RTTITable::retrieveMethodIndex when the class of the called method is not in RTTITable::\_classInfoTable.

* + method is a virtual method of the class containing the ClassInfo in construction,
  + \*methodIndex = 0, \*shiftObject = **nullptr**, \*shiftPvmt = **nullptr**.
  + The constructor MethodCall::MethodCall(clang::CXXMethodDecl\*, int64\_t\*, */\* inheritance \*/* list\*, */\* inheritance \*/* list\*) and the method MethodCall::add,
  + the method addFieldAccess,
  + the methods RTTITable::retrieveMethodIndex and Visitor::makeMemberCallExpression.

**void** addFieldAccess(clang::CXXRecordDecl\* baseClass, int64\_t\* accessIndex);

Record the accesses to a virtual base. It stores the triple methodIndex, shiftObject, shiftPvmt into a MethodCall attached to virtualBaseClass. This MethodCall is then stored in \_delayedCalls for the completion of accessIndex. When the ClassInfo is available the method updateWith will complete these arguments.

The implementation calls the constructor MethodCall::MethodCall(clang::CXXMethodDecl\*, int64\_t\*) or the method MethodCall::add.

This method is called by the method RTTITable::retrieveInheritancePathBetween, RTTITable::retrieveBaseIndex when the class of the called method is not in RTTITable::\_classInfoTable.

* + baseClass is a (virtual) base class of the class containing the ClassInfo in construction,
  + \*accessIndex = 0.
  + The constructor MethodCall::MethodCall(clang::CXXMethodDecl\*, int64\_t\*) and the method MethodCall::add,
  + the method addMethodCall,
  + the methods RTTITable::retrieveInheritancePathBetween, RTTITable::retrieveVirtualBaseIndex and Visitor::makeBaseToDerivedPointerCastExpression, Visitor::makeBaseToDerivedReferenceCastExpression, Visitor::makeDerivedToBasePointerCastExpression, Visitor::makeDerivedToBaseReferenceCastExpression, Visitor::insertConstructorPreambleIn.

**void** updateWith(**const** Clang\_utils& utils, **const** ClassInfo& methodTable);

Completes the fields that have been referenced in the range \_delayedCalls.begin()->\_indexShiftObject … \_delayedCalls.end()‑>\_indexShiftObject with the information contained in methodTable.

The implementation calls the method MethodCall::apply for all the elements of \_delayedCalls.

This method is called by the method RTTITable::exitClass each time a new ClassInfo has been completed with some DelayedMethodCalls waiting for it.

* + The method MethodCall::apply,
  + the methods ClassInfo::getIndexOfMethod, ClassInfo::getBasePosition,
  + the methods ClassInfo::getInheritancePath, ClassInfo::getVirtualInheritancePath,
  + the methods ClassInfo::hasPvmtAsFirstField, ClassInfo::getPvmtField,
  + the method RTTITable::exitClass.

The class RTTITable

The class RTTITable provides an interface to manage the inheritance graphs during the visit of the clang AST. Its main field RTTITable::\_classInfoTable is a map from clang C++ classes to local information of type ClassInfo. Hence, the class ClassInfo defines the information required to implement the following features at the level of each clang class:

* the generation of the structures for run-time type information (static structures, dynamic connection in the constructor/destructor of the classes and algorithms),
* the management of virtual method tables (creation and transformation of virtual method calls into an index in the right virtual method table),
* the translation of unambiguous but implicit inheritance paths into explicit inheritance paths,

During the visit of a class, it looks if a virtual method/a virtual base class is defined. If such a method/class is defined, either the first inherited class has virtual methods/virtual base classes and we reuse its Virtual Method Table for our class, or the first inherited class has no virtual method and we create a Virtual Method Table accessed by a new field pvmt (see the method addDerivation).

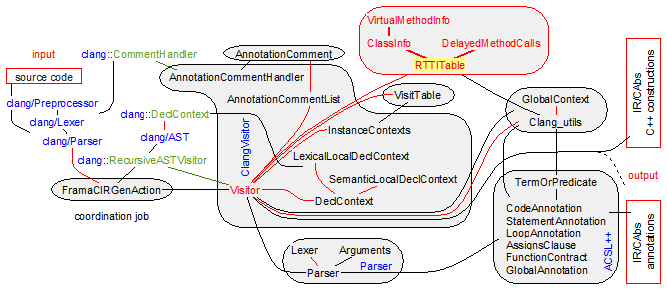
When a class has Virtual Method Table, our class is able to associate an index to each method declaration of type clang::CXXMethodDecl. The method retrieveMethodIndex is the interface method that does the job.

When a class has an inheritance graph, our class is able to retrieve the inheritance path between a derived class and an inherited class. The method retrieveInheritancePathBetween is the interface method that does the job.

When a constructor/destructor is visited, the C++ language does not authorize the call to virtual methods of more concrete but partially defined classes. To ensure this behavior, each constructor/destructor changes the access to the virtual method table with an access to its own virtual method table. As the visit of constructors may occur before the visit of the declaration of virtual methods, during the visit of constructors/destructors, our unit only registers the insertion points for the changes to the access to the virtual method table. Then at the end of the visit of the class, all virtual method declarations have been encountered and these insertion points are updated with the access to the right method table.

When a method call is visited, our unit may have no information about the method (whether it is virtual and its index in the table) because the method declaration may be not visited at that time. So we register all such undefined calls in \_delayedMethodCalls. When such calls are solved by the visit of the right declaration, the method DelayedMethodCalls::updateWith enables to unregister them.

The class environment of the class RTTITable is defined on the schema below:



Fields of the class RTTITable

**bool** \_hasInitRTTIFunctions;

Global mark to know if functions like **dynamic\_cast** have been generated. Such functions are generated with the first call to exitClass where the underlying \_currentClass.back() has got a virtual method table. Then \_hasInitRTTIFunctions becomes **true** and no other such generation can occur.

std::map<**const** clang::CXXRecordDecl\*, ClassInfo> \_classInfoTable;

Global association table that maps visited clang::CXXRecordDecl\* to their virtual method table. When the visitor enters a new class body (see the method enterClass), it updates the fields \_currentClass and \_currentClassInfo. During the visit, the virtual method table in \_currentClassInfo receives more and more information until it is complete. Then the method exitClass move this complete information into our field \_classInfoTable.

This field provides invaluable information on base classes, on classes that appear during the visit of a function/method (see the methods hasPvmt, hasVirtualMethods, getBasePosition, retrieveMethodIndex, retrieveBaseIndex, retrieveInheritancePathBetween.

std::list<**const** clang::CXXRecordDecl\*> \_currentClass;

This field represents the classes that are currently visited, as C++ authorizes nested classes. The first one \_currentClass.front() is the class under visit. It is set up by the method enterClass and it is erased by the method exitClass. The elements of \_currentClass are in connection with the virtual method tables in construction in \_currentClassInfo.

std::list<ClassInfo> \_currentClassInfo;

This field represents the virtual method tables under construction during the visit of the classes in \_currentClass. The first one \_currentClassInfo is the virtual method table under construction. This field is set up by the method enterClass and it is moved into \_classInfoTable by the method exitClass.

std::list<std::vector<*/\* statement \*/* list\*> > \_pvmtInsertionPoints;

During the visit of constructors/destructors, it is necessary to define the right method table for the calls to virtual methods. For a constructor, the policy is to call the inherited constructor, then to set up the method table with the table associated to the class of the constructor and at end to set up the fields and to execute the body of the constructor. As the virtual method table of a class is only known at the end of the visit, the constructors inside the class have no knowledge of it. So they generate the code in the intermediate format and they track the places where the access to the virtual method table should be inserted. These places are stored in \_pvmtInsertionPoints with the method addPvmtSetter(*/\* statement \*/* list&) (see the Visitor::insertConstructorPreambleIn and Visitor::insertDestructorPreambleIn) methods. When the entire virtual method table is known (see the method exitClass), the call to addPvmtSetter(**const** Clang\_utils&, **const** clang::CXXRecordDecl\*, */\* statement \*/* list&, location) inserts the access initialization of the virtual method table at each place in \_pvmtInsertionPoints.front().

Note that the elements of \_pvmtInsertionPoints are in connection with the nested class \_currentClass.

std::vector<**bool**\*> \_virtualTags;

List of pointers onto the field **struct** \_translation\_unit\_decl::cons\_translation\_unit\_decl.Compound.has\_virtual of the generation of the current class. As soon as the method addVirtualMethod is called, the field ….has\_virtual becomes **true**. This means that a virtual method table should be generated.

Note that the elements of \_pvmtInsertionPoints are in reverse connection with the nested class \_currentClass.

std::map<**const** clang::CXXRecordDecl\*, DelayedMethodCalls> \_delayedMethodCalls;

This field manages the calls to the virtual method of \_currentClass before this class is completely visited. As the entries in the virtual method table have not been built, in particular for the cast of the object (implicit first argument of the virtual method call), the simplest way to manage this delay is to register the calls into the field \_delayedMethodCalls (see the method retrieveMethodIndex/retrieveBaseIndex). At the end of the visit (see the method exitClass) \_currentClassInfo is completely built. Then the method DelayedMethodCalls::updateWith updates the generated calls with the full knowledge of the ClassInfo. Thanks to the mechanism of template class instantiation, this field is used in different situations and not only for nested classes. That is why it is not a list in connection with \_currentClass.

Declaration of the class RTTITable

**class** RTTITable {

**private**:

**bool** \_hasInitRTTIFunctions;

std::map<**const** clang::CXXRecordDecl\*, ClassInfo> \_classInfoTable;

std::list<**const** clang::CXXRecordDecl\*> \_currentClass;

std::list<ClassInfo> \_currentClassInfo;

std::list<std::vector<list\*> > \_pvmtInsertionPoints;

std::vector<**bool**\*> \_virtualTags;

std::map<**const** clang::CXXRecordDecl\*, DelayedMethodCalls> \_delayedMethodCalls;

**void** insertVMTContentPrelude(ForwardReferenceList& globals, location loc);

**void** insertVMTTypePrelude(ForwardReferenceList& globals, location loc);

**void** insertRTTIInfoPrelude(ForwardReferenceList& globals, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsAuxDeclPrelude(ForwardReferenceList& globals, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsAuxWhilePrelude(*/\* statement \*/* list\* functionBodyRef, location loc);

**void** insertRTTIAlgorithmsAuxWhileBodyPrelude(*/\* statement \*/* list\* whileBodyRef, location loc, */\* statement \*/* list\*& then, */\* statement \*/* list\*& elseThen, */\* statement \*/* list\*& elseElseThen);

**void** insertRTTIAlgorithmsAuxWhileThenBodyPrelude(*/\* statement \*/* list\* statements, location loc);

**void** insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude(*/\* statement \*/* list\* statements, location loc);

**void** insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude(*/\* statement \*/* list\* statements, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsAuxPrelude(ForwardReferenceList& globals, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsDeclPrelude(ForwardReferenceList& globals, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsFirstPartPrelude(*/\* statement \*/* list\* statements, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsDoWhilePrelude(*/\* statement \*/* list\*& statements, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsSecondWhilePrelude(*/\* statement \*/* list\*& statements, location loc);

**void** insertRTTIAlgorithmsSecondWhileBodyPrelude(*/\* statement \*/* list\*& statements, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsSecondIfThenPrelude(*/\* statement \*/* list\*& statements, location loc);

**void** insertRTTIAlgorithmsSecondIfThenBodyPrelude(*/\* statement \*/* list\*& statements, location loc);

**void** insertRTTIAlgorithmsLastPartPrelude(*/\* statement \*/* list\*& statements, location loc);

*/\* statement \*/* list\* insertRTTIAlgorithmsPrelude(ForwardReferenceList& globals, location loc);

**void** insertRttiPrelude(ForwardReferenceList& globals, location loc);

**void** insertStaticVMTDeclaration(ForwardReferenceList& classContent, location classLoc);

qualified\_name insertStaticVMTDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc);

**void** insertStaticBaseClassesDeclaration(ForwardReferenceList& classContent, location classLoc);

qualified\_name insertStaticBaseClassesDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc);

**void** insertStaticRTTIDeclaration(ForwardReferenceList& classContent, location classLoc);

qualified\_name insertStaticRTTIDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc, qualified\_name vmtName, qualified\_name nameBaseClasses);

**void** insertStaticVMTHeaderDeclaration(ForwardReferenceList& classContent, location classLoc);

**void** insertStaticVMTHeaderDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc, qualified\_name vmtName, qualified\_name nameRtti);

**bool** retrieveStaticInheritancePathBetween(**const** clang::CXXRecordDecl\* derived, **const** clang::CXXRecordDecl\* base, InheritancePath& result, **const** Clang\_utils& utils, VirtualInheritancePath\* virtualResult = **nullptr**) const;

**void** addBareToQualification(qualified\_name& name) **const**;

**public**:

RTTITable() : \_hasInitRTTIFunctions(**false**), \_currentClass(), \_currentClassInfo(), \_pvmtInsertionPoints(){}

**bool** hasDelayedMethodCalls() **const** { **return** !\_delayedMethodCalls.empty(); }

**void** insertVMTAndRttiPrelude(ForwardReferenceList& globals, location loc);

**void** enterClass(**const** clang::CXXRecordDecl\* classDecl)

{ \_currentClass.push\_front(classDecl); \_currentClassInfo.push\_front(ClassInfo()); \_pvmtInsertionPoints.push\_front(std::vector<list\*>()); }

**const** clang::CXXRecordDecl\* getCurrentClass() **const** { **return** \_currentClass.empty() ? **nullptr** : \_currentClass.front(); }

**void** exitClass() { \_currentClass.pop\_front(); \_currentClassInfo.pop\_front(); \_pvmtInsertionPoints.pop\_front(); }

**void** exitClass(**const** Clang\_utils& utils, ForwardReferenceList& content, ForwardReferenceList& globals, location classLoc);

expression getPvmt(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* source, expression arg) **const**;

**void** addPvmtSetter(*/\* statement \*/* list& insertionPoint) { \_pvmtInsertionPoints.front().push\_back(&insertionPoint); }

**void** addPvmtSetter(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* source, */\* statement \*/* list& insertionPoint, location loc);

**int** addDerivation(**const** clang::CXXRecordDecl\* source, **bool** isVirtual);

**bool** hasVirtualMethods() **const** { **return** \_currentClassInfo.front().numberOfMethods() > 0; }

**bool** hasVirtualMethods(**const** clang::CXXRecordDecl\* source) **const**

{ **auto** found = \_classInfoTable.find(source); **return** (found != \_classInfoTable.end()) && found->second.numberOfMethods() > 0; }

**const** ClassInfo\* getClassInfo(**const** clang::CXXRecordDecl\* source) **const**;

**bool** hasVirtualBaseClasses(**const** clang::CXXRecordDecl\* source) **const**

{ **auto** found = \_classInfoTable.find(source); **return** (found != \_classInfoTable.end()) && found->second.hasVirtualBaseClasses(); }

**bool** hasPvmt(**const** clang::CXXRecordDecl\* source) **const**

{ **auto** found = \_classInfoTable.find(source); **return** (found != \_classInfoTable.end()) && !found->second.isEmpty(); }

**int** getBasePosition(**const** clang::CXXRecordDecl\* derived, **const** clang::CXXRecordDecl\* base, **bool**& isVirtual) **const**;

**void** addVirtualMethod(clang::CXXMethodDecl\* method)

{ \_currentClassInfo.front().addVirtualMethod(method, \_currentClass.front()); \*\_virtualTags.back() = **true**; }

**void** setVirtualTag(**bool**& virtualTag) { \_virtualTags.push\_back(&virtualTag); }

**void** retrieveMethodIndex(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* classCaller, clang::CXXMethodDecl\* methodCalled, int64\_t\* methodIndex, */\* inheritance \*/* list\* shiftObject, */\* inheritance \*/* list\* shiftPvmt);

**void** retrieveBaseIndex(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* classCaller, clang::CXXRecordDecl\* baseClass, int64\_t\* accessIndex);

**void** retrieveInheritancePathBetween(**const** clang::CXXRecordDecl\* derived, **const** clang::CXXRecordDecl\* base, InheritancePath& result, VirtualInheritancePath& virtualResult, **const** Clang\_utils& utils) **const**;

};

Methods of the class RTTITable

Private methods

**void** insertVMTContentPrelude(ForwardReferenceList& globals, location loc);

Generate the following definition at the beginning of each translation unit:

**struct** \_frama\_c\_vmt\_content {

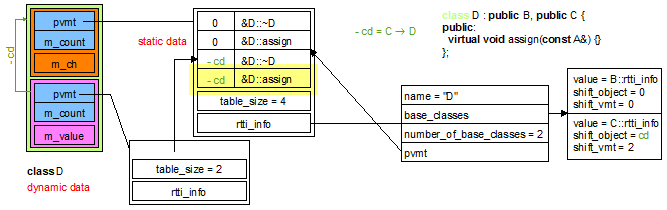
**void** (\*method\_ptr)();

**int** shift\_this;

};

It defines the cell in the virtual method table enabling to call a virtual method.

For the example 1, this data structure is underlined in yellow on the following schema:



This method is called by the method insertVMTAndRttiPrelude where a more complete description is available.

* + The method insertVMTAndRttiPrelude,
  + the methods insertVMTTypePrelude, insertRTTIInfoPrelude,
  + the constructor Visitor::Visitor(FILE\*, clang::CompilerInstance&).

**void** insertVMTTypePrelude(ForwardReferenceList& globals, location loc);

Generate the following definitions at the beginning of each translation unit:

**struct** \_frama\_c\_vmt {

**struct** \_frama\_c\_vmt\_content\* table;

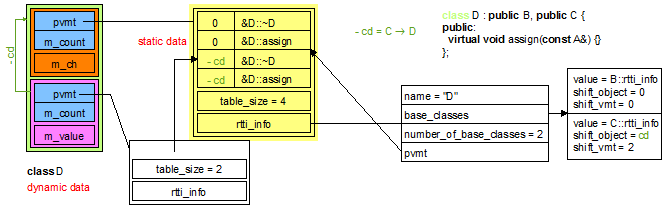
**int** table\_size;

**struct** \_frama\_c\_rtti\_name\_info\_node\* rtti\_info;

};

It defines the data structure for the virtual method table and for the root of the run-time type information (or rtti for short). The data structure for the body of the rtti is given by the method insertRTTIInfoPrelude. A pointer on **struct** \_frama\_c\_vmt is systematically defined in each generated object of a class that has virtual methods or virtual base classes.

For the example 1, this data structure is underlined in yellow on the following schema:



This method is called by the method insertVMTAndRttiPrelude where a more complete description is available.

* + The method insertVMTAndRttiPrelude,
  + the methods insertVMTContentPrelude, insertRTTIInfoPrelude,
  + the constructor Visitor::Visitor(FILE\*, clang::CompilerInstance&).

**void** insertRTTIInfoPrelude(ForwardReferenceList& globals, location loc);

Generate the following definitions at the beginning of each translation unit:

**struct** \_frama\_c\_rtti\_name\_info\_content {

**struct** \_frama\_c\_rtti\_name\_info\_node\* value;

**int** shift\_object;

**int** shift\_vmt;

};

**struct** \_frama\_c\_rtti\_name\_info\_node {

**const** **char**\* name;

**struct** \_frama\_c\_rtti\_name\_info\_content\* base\_classes;

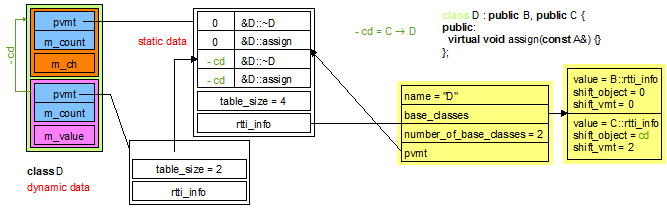
**int** number\_of\_base\_classes;

**struct** \_frama\_c\_vmt\* pvmt;

};

It defines the recursive data structure for the internal part of the run-time type information (or rtti for short). This internal part is accessible from the **struct** \_frama\_c\_vmt defined by insertVMTTypePrelude.

For the example 1, this data structure is underlined in yellow on the following schema:



The implementation calls the methods insertVMTContentPrelude, insertVMTTypePrelude, insertRTTIInfoPrelude.

It is called by the constructor of the class Visitor as the first code generation.

* + The methods insertVMTContentPrelude, insertVMTTypePrelude, insertRTTIInfoPrelude,
  + the constructor Visitor::Visitor(FILE\*, clang::CompilerInstance&).

*/\* statement \*/* list\* insertRTTIAlgorithmsAuxDeclPrelude(ForwardReferenceList& globals, location loc);

Generate in globals the external part of the function \_frama\_c\_find\_dynamic\_cast\_aux that defines the recursive part of the **dynamic\_cast** algorithm. The result is the place where new instructions can be inserted in the function’s body. Our method is called by the method insertRttiPrelude.

**int** \_frama\_c\_find\_dynamic\_cast\_aux(**struct** \_frama\_c\_rtti\_name\_info\_node\* target\_info, **struct** \_frama\_c\_rtti\_name\_info\_content\* concrete\_base, **int** number\_of\_bases, **int** found\_shift\_object, **int** found\_shift\_vmt, **int** last\_shift\_vmt, **int**\* shift\_object, **int**\* distance)

{ …

}

A more precise description is available at the documentation of the method insertRTTIAlgorithmsAuxPrelude that calls our method.

* + The methods insertRTTIAlgorithmsAuxWhilePrelude, insertRTTIAlgorithmsAuxWhileBodyPrelude, insertRTTIAlgorithmsAuxWhileThenBodyPrelude, insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude,
  + the method insertRTTIAlgorithmsAuxPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsAuxWhilePrelude(*/\* statement \*/* list\* functionBodyRef, location loc);

Generate at the place functionBodyRef some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast\_aux that defines the recursive part of the **dynamic\_cast** algorithm. The result is the place where new instructions can be inserted in the main “while” of the function’s body. Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsAuxPrelude that calls our method.

* + The methods insertRTTIAlgorithmsAuxDeclPrelude, insertRTTIAlgorithmsAuxWhileBodyPrelude, insertRTTIAlgorithmsAuxWhileThenBodyPrelude, insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude,
  + the method insertRTTIAlgorithmsAuxPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

**void** insertRTTIAlgorithmsAuxWhileBodyPrelude(*/\* statement \*/* list\* whileBodyRef, location loc, */\* statement \*/* list\*& then, */\* statement \*/* list\*& elseThen, */\* statement \*/* list\*& elseElseThen);

Generate at the place whileBodyRef some instructions of the main “while” of the function’s body of \_frama\_c\_find\_dynamic\_cast\_aux that defines the recursive part of the **dynamic\_cast** algorithm. The results then, elseThen, elseElseThen are the places where new instructions can be inserted. Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsAuxPrelude that calls our method.

* + The methods insertRTTIAlgorithmsAuxDeclPrelude, insertRTTIAlgorithmsAuxWhilePrelude, insertRTTIAlgorithmsAuxWhileThenBodyPrelude, insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude,
  + the method insertRTTIAlgorithmsAuxPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

**void** insertRTTIAlgorithmsAuxWhileThenBodyPrelude(*/\* statement \*/* list\* statements, location loc);

Generate at the place statements some instructions of the “then” branch of the “if” in the main “while” of the function’s body of \_frama\_c\_find\_dynamic\_cast\_aux that defines the recursive part of the **dynamic\_cast** algorithm (see the argument then of the method insertRTTIAlgorithmsAuxWhileBodyPrelude). Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsAuxPrelude that calls our method.

* + The methods insertRTTIAlgorithmsAuxDeclPrelude, insertRTTIAlgorithmsAuxWhilePrelude, insertRTTIAlgorithmsAuxWhileBodyPrelude, insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude,
  + the method insertRTTIAlgorithmsAuxPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

**void** insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude(*/\* statement \*/* list\* statements, location loc);

Generate at the place statements some instructions of the “then” branch of the “if” in the “else” branch of the “if” in the main “while” of the function’s body of \_frama\_c\_find\_dynamic\_cast\_aux that defines the recursive part of the **dynamic\_cast** algorithm (see the argument elseThen of the method insertRTTIAlgorithmsAuxWhileBodyPrelude). Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsAuxPrelude that calls our method.

* + The methods insertRTTIAlgorithmsAuxDeclPrelude, insertRTTIAlgorithmsAuxWhilePrelude, insertRTTIAlgorithmsAuxWhileBodyPrelude, insertRTTIAlgorithmsAuxWhileThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude,
  + the method insertRTTIAlgorithmsAuxPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

**void** insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude(*/\* statement \*/* list\* statements, location loc);

Generate at the place statements some instructions of the “then” branch of the second “if” in the “else” branch of the “if” in the main “while” of the function’s body of \_frama\_c\_find\_dynamic\_cast\_aux that defines the recursive part of the **dynamic\_cast** algorithm (see the argument elseElseThen of the method insertRTTIAlgorithmsAuxWhileBodyPrelude). Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsAuxPrelude that calls our method.

* + The methods insertRTTIAlgorithmsAuxDeclPrelude, insertRTTIAlgorithmsAuxWhilePrelude, insertRTTIAlgorithmsAuxWhileBodyPrelude, insertRTTIAlgorithmsAuxWhileThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude,
  + the method insertRTTIAlgorithmsAuxPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsAuxPrelude(ForwardReferenceList& globals, location loc);

Generate in globals the function \_frama\_c\_find\_dynamic\_cast\_aux that defines the recursive part of the **dynamic\_cast** algorithm. The result is the place where new instructions can be inserted in the function’s body, after this generation. Our method is called by the method insertRttiPrelude.

*/\* generated by insertRTTIAlgorithmsAuxDeclPrelude \*/*

**int** \_frama\_c\_find\_dynamic\_cast\_aux(**struct** \_frama\_c\_rtti\_name\_info\_node\* target\_info, **struct** \_frama\_c\_rtti\_name\_info\_content\* concrete\_base, **int** number\_of\_bases, **int** found\_shift\_object, **int** found\_shift\_vmt, **int** last\_shift\_vmt, **int**\* shift\_object, **int**\* distance)

{ */\* generated by insertRTTIAlgorithmsAuxWhilePrelude \*/*

**int** result = 0;

**struct** \_frama\_c\_rtti\_name\_info\_content\* cursor = concrete\_base;

**int** is\_over = 0;

**int** base\_index = 0;

**while** (base\_index < number\_of\_bases) {

*/\* generated by insertRTTIAlgorithmsAuxWhileBodyPrelude \*/*

**if** (cursor->value == target\_info) {

**if** (\*distance < 0 || \*distance >= 1) {

*/\* generated by insertRTTIAlgorithmsAuxWhileThenBodyPrelude \*/*

*/\* case 0 and 1 are not possible \*/*

**if** (found\_shift\_vmt == cursor->shift\_vmt)

\*distance = 0; */\* impossible case \*/*

**else**

\*distance = 1;

\*shift\_object = found\_shift\_object - cursor->shift\_object;

result = 1;

} */\* generated by insertRTTIAlgorithmsAuxWhileBodyPrelude \*/*

}

**else** **if** (cursor->shift\_vmt <= found\_shift\_vmt && ((base\_index < number\_of\_bases-1)

? ((cursor+1)->shift\_vmt > found\_shift\_vmt)

: ((last\_shift\_vmt == -1) || (found\_shift\_vmt < last\_shift\_vmt))))

{ */\* generated by insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude \*/*

**int** local\_distance = 0;

**int** local\_shift\_object = 0;

**int** local\_result = \_frama\_c\_find\_dynamic\_cast\_aux(target\_info, cursor->value->base\_classes, cursor->value->number\_of\_base\_classes, found\_shift\_object - cursor->shift\_object, found\_shift\_vmt - cursor->shift\_vmt,

(base\_index < number\_of\_bases-1) ? (cursor+1)->shift\_vmt : last\_shift\_vmt, &local\_shift\_object,

&local\_distance);

**if** (local\_result && (local\_distance >= 0)) {

**if** (\*distance < 0 || \*distance >= local\_distance) {

result = local\_result;

\*shift\_object = local\_shift\_object - cursor->shift\_object;

\*distance = local\_distance;

};

};

is\_over = 1;

} */\* generated by insertRTTIAlgorithmsAuxWhileBodyPrelude \*/*

**else** **if** (\*distance < 0 || \*distance > 1) {

*/\* generated by insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude \*/*

**int** local\_distance = 0;

**int** local\_shift\_object = 0;

**int** local\_result = \_frama\_c\_find\_dynamic\_cast\_aux(target\_info, cursor->value->base\_classes, cursor->value->number\_of\_base\_classes, found\_shift\_object - cursor->shift\_object, found\_shift\_vmt - cursor->shift\_vmt,

(base\_index < number\_of\_bases-1) ? (cursor+1)->shift\_vmt : last\_shift\_vmt,

&local\_shift\_object, &local\_distance);

**if** (local\_result && (local\_distance >= 0)) {

**if** (\*distance < 0 || \*distance > (!is\_over ? local\_distance : (local\_distance+1))) {

result = local\_result;

\*shift\_object = local\_shift\_object - cursor->shift\_object;

\*distance = local\_distance+1;

};

};

}; */\* generated by insertRTTIAlgorithmsAuxWhileBodyPrelude \*/*

cursor = cursor+1;

base\_index = base\_index+1;

}; */\* end of while \*/ /\* generated by insertRTTIAlgorithmsAuxWhilePrelude \*/*

**return** result;

}

The implementation simply calls the methods insertRTTIAlgorithmsAuxDeclPrelude, insertRTTIAlgorithmsAuxWhilePrelude, insertRTTIAlgorithmsAuxWhileBodyPrelude, insertRTTIAlgorithmsAuxWhileThenBodyPrelude, insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude.

* + The methods insertRTTIAlgorithmsAuxDeclPrelude, insertRTTIAlgorithmsAuxWhilePrelude, insertRTTIAlgorithmsAuxWhileBodyPrelude, insertRTTIAlgorithmsAuxWhileThenBodyPrelude, insertRTTIAlgorithmsAuxWhileElseThenBodyPrelude and insertRTTIAlgorithmsAuxWhileElseElseThenBodyPrelude,
  + the methods insertRttiPrelude and insertRTTIAlgorithmsPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsDeclPrelude(ForwardReferenceList& globals, location loc);

Generate in globals the external part of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The result is the place where new instructions can be inserted in the function’s body, after this generation. Our method is called by the method insertRttiPrelude.

**int** \_frama\_c\_find\_dynamic\_cast(**struct** \_frama\_c\_rtti\_name\_info\_node\* declared\_info, **struct** \_frama\_c\_vmt\* declared\_vmt, **struct** \_frama\_c\_rtti\_name\_info\_node\* target\_info, **int**\* shift\_object)

{ …

}

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsFirstPartPrelude(*/\* statement \*/* list\* statements, location loc);

Generate at the place statements some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The result is the place where new instructions can be inserted after the generated instructions in the function’s body. Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsDoWhilePrelude(*/\* statement \*/* list\*& statements, location loc);

Generate at the place statements some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The result is the place where new instructions can be inserted inside the “do { … } while” after the generated instructions in the function’s body. statements advances with the newly inserted instructions. Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsSecondWhilePrelude(*/\* statement \*/* list\*& statements, location loc);

Generate at the place statements some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The result is the place where new instructions can be inserted inside the “while” after the generated instructions in the “do { … } while” (see the result of insertRTTIAlgorithmsDoWhilePrelude). statements advances with the newly inserted instructions. Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

**void** insertRTTIAlgorithmsSecondWhileBodyPrelude(*/\* statement \*/* list\*& statements, location loc);

Generate at the place statements some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The newly inserted instructions are inside the “while” after the generated instructions in the “do { … } while” (see the result of insertRTTIAlgorithmsSecondWhilePrelude). Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondIfThenPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsSecondIfThenPrelude(*/\* statement \*/* list\*& statements, location loc);

Generate at the place statements some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The newly inserted instructions are a conditional after the “while” after the generated instructions in the “do { … } while” (see the advance of the argument statement of insertRTTIAlgorithmsSecondWhilePrelude). The result is the place where new instructions can be inserted inside the “then” branch of the conditional. Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

**void** insertRTTIAlgorithmsSecondIfThenBodyPrelude(*/\* statement \*/* list\*& statements, location loc);

Generate at the place statements some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The newly inserted instructions are inside the “then” branch of the conditional after the “while” after the generated instructions in the “do { … } while” (see the result of insertRTTIAlgorithmsSecondIfThenPrelude). Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

**void** insertRTTIAlgorithmsLastPartPrelude(*/\* statement \*/* list\*& statements, location loc);

Generate at the place statements some instructions of the body of the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. The newly inserted instructions are inside the “then” branch of the conditional after the “do { … } while” (see the advance of the argument statements of insertRTTIAlgorithmsDoWhilePrelude). Our method is called by the method insertRttiPrelude.

A more precise description is available at the documentation of the method insertRTTIAlgorithmsPrelude that calls our method.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude and insertRTTIAlgorithmsSecondIfThenBodyPrelude,
  + the method insertRTTIAlgorithmsPrelude,
  + the method insertRttiPrelude,
  + the method exitClass.

*/\* statement \*/* list\* insertRTTIAlgorithmsPrelude(ForwardReferenceList& globals, location loc);

Generate in globals the function \_frama\_c\_find\_dynamic\_cast that defines the **dynamic\_cast** algorithm. Note that this function calls the recursive function \_frama\_c\_find\_dynamic\_cast\_aux. The result is the place where new instructions can be inserted in the function’s body, after this generation. Our method is called by the method insertRttiPrelude.

*/\* generated by insertRTTIAlgorithmsDeclPrelude \*/*

**int** \_frama\_c\_find\_dynamic\_cast(**struct** \_frama\_c\_rtti\_name\_info\_node\* declared\_info, **struct** \_frama\_c\_vmt\* declared\_vmt, **struct** \_frama\_c\_rtti\_name\_info\_node\* target\_info, **int**\* shift\_object)

{ */\* generated by insertRTTIAlgorithmsFirstPartPrelude \*/*

**struct** \_frama\_c\_rtti\_name\_info\_node\* concrete\_info = declared\_vmt->rtti\_info;

**int** shift\_vmt, elaborated\_shift\_vmt = 0, elaborated\_shift\_object = 0, elaborated\_shift\_target;

**struct** \_frama\_c\_rtti\_name\_info\_content\* cursor;

**int** cursor\_index = 0, number\_of\_bases;

**int** distance = -1;

**if** (concrete\_info->pvmt > declared\_vmt || declared\_vmt > concrete\_info->pvmt + declared\_vmt->table\_size)

**return** 0;

shift\_vmt = declared\_vmt - concrete\_info->pvmt;

*// 0 <= shift\_vmt <= table\_size*

**if** (concrete\_info == declared\_info) {

\*shift\_object = 0;

**return** (target\_info == declared\_info);

};

elaborated\_shift\_target = (target\_info == concrete\_info) ? 0 : -1;

cursor = concrete\_info->base\_classes;

number\_of\_bases = concrete\_info->number\_of\_base\_classes;

*/\* generated by insertRTTIAlgorithmsDoWhilePrelude \*/*

**do** {

*/\* generated by insertRTTIAlgorithmsSecondWhilePrelude \*/*

**while** ((cursor\_index < number\_of\_bases) && (elaborated\_shift\_vmt+cursor->shift\_vmt < shift\_vmt))

{ */\* generated by insertRTTIAlgorithmsSecondWhileBodyPrelude \*/*

**struct** \_frama\_c\_rtti\_name\_info\_content\* next\_cursor

= (cursor\_index < number\_of\_bases-1) ? (cursor+1) : (struct \_frama\_c\_rtti\_name\_info\_content\*) 0;

**if** (next\_cursor && (elaborated\_shift\_vmt + next\_cursor->shift\_vmt <= shift\_vmt)) {

cursor = next\_cursor;

cursor\_index = cursor\_index + 1;

}

**else**

**break**;

}; */\* generated by insertRTTIAlgorithmsSecondWhilePrelude \*/*

*/\* generated by insertRTTIAlgorithmsSecondIfThenPrelude \*/*

**if** (cursor\_index < number\_of\_bases) {

*/\* generated by insertRTTIAlgorithmsSecondIfThenBodyPrelude \*/*

elaborated\_shift\_vmt += cursor->shift\_vmt;

elaborated\_shift\_object += cursor->shift\_object;

**if** (cursor->value == target\_info)

elaborated\_shift\_target = elaborated\_shift\_object;

**if** (elaborated\_shift\_vmt == shift\_vmt && cursor->value == declared\_info) {

**if** (elaborated\_shift\_target >= 0) {

\*shift\_object = elaborated\_shift\_target-elaborated\_shift\_object;

**return** 1;

};

**break**;

};

cursor = cursor->value->base\_classes;

number\_of\_bases = cursor->value->number\_of\_base\_classes;

cursor\_index = 0;

}; */\* generated by insertRTTIAlgorithmsSecondIfThenPrelude \*/*

} **while** (cursor\_index < number\_of\_bases); */\* generated by insertRTTIAlgorithmsDoWhilePrelude \*/*

*/\* generated by insertRTTIAlgorithmsLastPartPrelude \*/*

**if** (cursor\_index >= number\_of\_bases)

**return** 0;

*// elaborated\_shift\_target == -1 && cursor->value == target\_info && elaborated\_shift\_vmt == shift\_vmt*

**return** \_frama\_c\_find\_dynamic\_cast\_aux(target\_info, concrete\_info->base\_classes,

concrete\_info->number\_of\_base\_classes, elaborated\_shift\_object, shift\_vmt, -1, shift\_object, &distance);

}

The implementation simply calls the methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude.

* + The methods insertRTTIAlgorithmsDeclPrelude, insertRTTIAlgorithmsFirstPartPrelude, insertRTTIAlgorithmsDoWhilePrelude, insertRTTIAlgorithmsSecondWhilePrelude, insertRTTIAlgorithmsSecondWhileBodyPrelude, insertRTTIAlgorithmsSecondIfThenPrelude, insertRTTIAlgorithmsSecondIfThenBodyPrelude and insertRTTIAlgorithmsLastPartPrelude,
  + the methods insertRttiPrelude and insertRTTIAlgorithmsAuxPrelude,
  + the method exitClass.

**void** insertRttiPrelude(ForwardReferenceList& globals, location loc);

Generate the **dynamic\_cast** algorithms as soon as a virtual method or a virtual base class is visited (see the method exitClass). This generation is controlled by the flag \_hasInitRTTIFunctions, initially **false** and then **true** just after the generation.

*/\* generated by insertRTTIAlgorithmsAuxPrelude \*/*

**int** \_frama\_c\_find\_dynamic\_cast\_aux(**struct** \_frama\_c\_rtti\_name\_info\_node\* target\_info, **struct** \_frama\_c\_rtti\_name\_info\_content\* concrete\_base, **int** number\_of\_bases, **int** found\_shift\_object, **int** found\_shift\_vmt, **int** last\_shift\_vmt, **int**\* shift\_object, **int**\* distance)

{ **int** result = 0;

**struct** \_frama\_c\_rtti\_name\_info\_content\* cursor = concrete\_base;

**int** is\_over = 0;

**int** base\_index = 0;

**while** (base\_index < number\_of\_bases) {

**if** (cursor->value == target\_info) {

**if** (\*distance < 0 || \*distance >= 1) {

*/\* case 0 and 1 are not possible \*/*

**if** (found\_shift\_vmt == cursor->shift\_vmt)

\*distance = 0; */\* impossible case \*/*

**else**

\*distance = 1;

\*shift\_object = found\_shift\_object - cursor->shift\_object;

result = 1;

}

}

**else** **if** (cursor->shift\_vmt <= found\_shift\_vmt && ((base\_index < number\_of\_bases-1)

? ((cursor+1)->shift\_vmt > found\_shift\_vmt)

: ((last\_shift\_vmt == -1) || (found\_shift\_vmt < last\_shift\_vmt))))

{ **int** local\_distance = 0;

**int** local\_shift\_object = 0;

**int** local\_result = \_frama\_c\_find\_dynamic\_cast\_aux(target\_info, cursor->value->base\_classes, cursor->value->number\_of\_base\_classes, found\_shift\_object - cursor->shift\_object, found\_shift\_vmt - cursor->shift\_vmt,

(base\_index < number\_of\_bases-1) ? (cursor+1)->shift\_vmt : last\_shift\_vmt, &local\_shift\_object,

&local\_distance);

**if** (local\_result && (local\_distance >= 0)) {

**if** (\*distance < 0 || \*distance >= local\_distance) {

result = local\_result;

\*shift\_object = local\_shift\_object - cursor->shift\_object;

\*distance = local\_distance;

};

};

is\_over = 1;

}

**else** **if** (\*distance < 0 || \*distance > 1) {

**int** local\_distance = 0;

**int** local\_shift\_object = 0;

**int** local\_result = \_frama\_c\_find\_dynamic\_cast\_aux(target\_info, cursor->value->base\_classes, cursor->value->number\_of\_base\_classes, found\_shift\_object - cursor->shift\_object, found\_shift\_vmt - cursor->shift\_vmt,

(base\_index < number\_of\_bases-1) ? (cursor+1)->shift\_vmt : last\_shift\_vmt,

&local\_shift\_object, &local\_distance);

**if** (local\_result && (local\_distance >= 0)) {

**if** (\*distance < 0 || \*distance > (!is\_over ? local\_distance : (local\_distance+1))) {

result = local\_result;

\*shift\_object = local\_shift\_object - cursor->shift\_object;

\*distance = local\_distance+1;

};

};

};

cursor = cursor+1;

base\_index = base\_index+1;

}; */\* end of while \*/*

**return** result;

}

*/\* generated by insertRTTIAlgorithmsPrelude \*/*

**int** \_frama\_c\_find\_dynamic\_cast(**struct** \_frama\_c\_rtti\_name\_info\_node\* declared\_info, **struct** \_frama\_c\_vmt\* declared\_vmt, **struct** \_frama\_c\_rtti\_name\_info\_node\* target\_info, **int**\* shift\_object)

{ **struct** \_frama\_c\_rtti\_name\_info\_node\* concrete\_info = declared\_vmt->rtti\_info;

**int** shift\_vmt, elaborated\_shift\_vmt = 0, elaborated\_shift\_object = 0, elaborated\_shift\_target;

**struct** \_frama\_c\_rtti\_name\_info\_content\* cursor;

**int** cursor\_index = 0, number\_of\_bases;

**int** distance = -1;

**if** (concrete\_info->pvmt > declared\_vmt || declared\_vmt > concrete\_info->pvmt + declared\_vmt->table\_size)

**return** 0;

shift\_vmt = declared\_vmt - concrete\_info->pvmt;

*// 0 <= shift\_vmt <= table\_size*

**if** (concrete\_info == declared\_info) {

\*shift\_object = 0;

**return** (target\_info == declared\_info);

};

elaborated\_shift\_target = (target\_info == concrete\_info) ? 0 : -1;

cursor = concrete\_info->base\_classes;

number\_of\_bases = concrete\_info->number\_of\_base\_classes;

**do** {

**while** ((cursor\_index < number\_of\_bases) && (elaborated\_shift\_vmt+cursor->shift\_vmt < shift\_vmt))

{ **struct** \_frama\_c\_rtti\_name\_info\_content\* next\_cursor

= (cursor\_index < number\_of\_bases-1) ? (cursor+1) : (struct \_frama\_c\_rtti\_name\_info\_content\*) 0;

**if** (next\_cursor && (elaborated\_shift\_vmt + next\_cursor->shift\_vmt <= shift\_vmt)) {

cursor = next\_cursor;

cursor\_index = cursor\_index + 1;

}

**else**

**break**;

};

**if** (cursor\_index < number\_of\_bases) {

elaborated\_shift\_vmt += cursor->shift\_vmt;

elaborated\_shift\_object += cursor->shift\_object;

**if** (cursor->value == target\_info)

elaborated\_shift\_target = elaborated\_shift\_object;

**if** (elaborated\_shift\_vmt == shift\_vmt && cursor->value == declared\_info) {

**if** (elaborated\_shift\_target >= 0) {

\*shift\_object = elaborated\_shift\_target-elaborated\_shift\_object;

**return** 1;

};

**break**;

};

cursor = cursor->value->base\_classes;

number\_of\_bases = cursor->value->number\_of\_base\_classes;

cursor\_index = 0;

};

} **while** (cursor\_index < number\_of\_bases);

**if** (cursor\_index >= number\_of\_bases)

**return** 0;

*// elaborated\_shift\_target == -1 && cursor->value == target\_info && elaborated\_shift\_vmt == shift\_vmt*

**return** \_frama\_c\_find\_dynamic\_cast\_aux(target\_info, concrete\_info->base\_classes,

concrete\_info->number\_of\_base\_classes, elaborated\_shift\_object, shift\_vmt, -1, shift\_object, &distance);

}

The implementation simply calls the methods insertRTTIAlgorithmsAuxPrelude and insertRTTIAlgorithmsPrelude.

* + The methods insertRTTIAlgorithmsAuxPrelude and insertRTTIAlgorithmsPrelude,
  + the method exitClass.

**void** insertStaticVMTDeclaration(ForwardReferenceList& classContent, location classLoc);

Insert the declaration

**struct** \_frama\_c\_vmt\_content \_frama\_c\_vmt[xxx];

to declare the virtual method table of the current class (see the field \_currentClassInfo.front()).

xxx stands for \_currentClassInfo.front().numberOfMethods().

Its definition is defined by the method insertStaticVMTDefinition. Our method is called by the method exitClass, before the methods insertStaticBaseClassesDeclaration, insertStaticRTTIDeclaration, insertStaticRTTIDeclaration and insertStaticVMTHeaderDeclaration for each class that has virtual methods.

* + The method insertStaticVMTDefinition,
  + the method exitClass,
  + the methods insertStaticBaseClassesDeclaration, insertStaticRTTIDeclaration, insertStaticVMTHeaderDeclaration.

qualified\_name insertStaticVMTDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc);

Insert the definition of the virtual method table of the current class (see the field \_currentClassInfo.front()) at the globals level.

**struct** \_frama\_c\_vmt\_content XXX::\_frama\_c\_vmt[xxx] = { {…, …}, …, {…, …} };

where XXX stands for makeQualifiedName(\_currentClass.front()) and xxx stands for \_currentClassInfo.front().numberOfMethods(); this definition is compatible with the declaration done by the method insertStaticVMTDeclaration.

For each 0 ≤ i < xxx, iter of type ClassInfo::MethodIterator points onto the ith element of the virtual method table (see \_currentClassInfo.front().beginOfMethods() and \_currentClassInfo.front().endOfMethods()).

* if iter->isMethod(),
* \_frama\_c\_vmt[i].method\_ptr = (void (\*)()) &makeQualifiedName(\*iter->getMethod());
* \_frama\_c\_vmt\_content[i].shift\_this

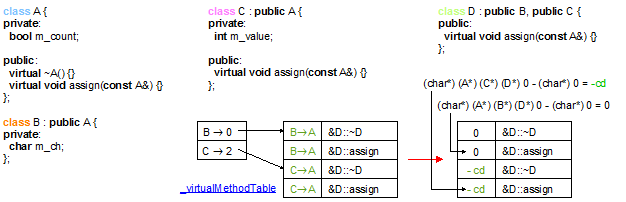
= (void\*) (((\_currentClass\*) 0)->\_frama\_c\_sub target) - (void\*) (((\_currentClass\*) 0)->\_frama\_c\_sub iter)

* if iter->isVirtualBaseClass(),
* \_frama\_c\_vmt[i].method\_ptr = (void (\*)()) 0;
* \_frama\_c\_vmt\_content[i].shift\_this

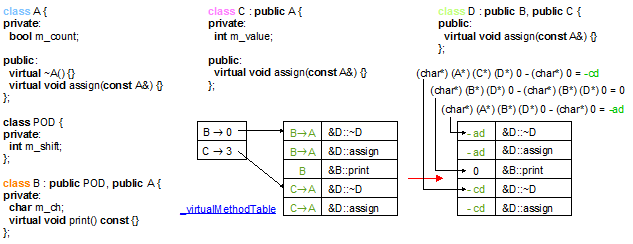
= (void\*) (((\_currentClass\*) 0)->\_frama\_c\_sub target) - (void\*) (((\_currentClass\*) 0)->\_frama\_c\_sub iter)

where target is the target of the class path iter->getVirtualInheritancePath()/iter->getInheritancePath().

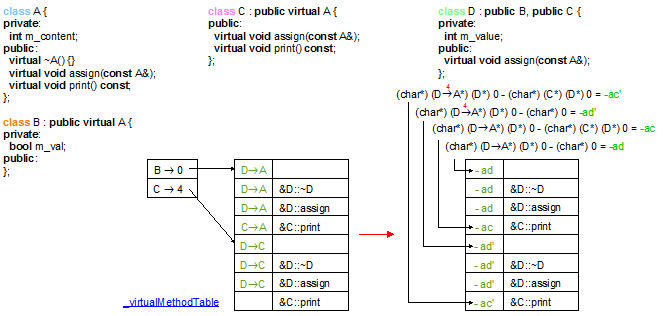
For the example 1,



For the example 2,



For the example 3,



Our method is called by the method exitClass, before the methods insertStaticBaseClassesDefinition, insertDeclStaticVMTHeaderDefinition, insertStaticRTTIDefinition and insertStaticVMTHeaderDefinition for each class that has virtual methods.

* + The method insertStaticVMTDeclaration,
  + the classes ClassInfo, ClassInfo::MethodIterator and ClassInfo::VirtualMethodInfo,
  + the method exitClass,
  + the methods the methods insertStaticBaseClassesDefinition, insertDeclStaticVMTHeaderDefinition, insertStaticRTTIDefinition and insertStaticVMTHeaderDefinition.

**void** insertStaticBaseClassesDeclaration(ForwardReferenceList& classContent, location classLoc);

Insert the declaration

**struct** \_frama\_c\_rtti\_name\_info\_content \_frama\_c\_base\_classes[xxx];

to declare the base classes of the current class (see the field \_currentClassInfo.front()). This field is required for the **dynamic\_cast** algorithm (see the method insertRTTIAlgorithmsFirstPartPrelude as example).

xxx is \_currentClassInfo.front().\_virtualDerivationsPosition.size() + \_currentClassInfo.front().\_derivationsPosition.size().

Its definition is defined by the method insertStaticBaseClassesDefinition. Our method is called by the method exitClass, between the method insertStaticVMTDeclaration and the methods insertStaticRTTIDeclaration, insertStaticRTTIDeclaration, insertStaticVMTHeaderDeclaration for each class that has virtual methods.

* + The method insertStaticBaseClassesDefinition,
  + the method exitClass,
  + the methods insertStaticVMTDeclaration, insertStaticRTTIDeclaration, insertStaticVMTHeaderDeclaration.

qualified\_name insertStaticBaseClassesDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc);

Insert the definition of the base classes of the current class (see the field \_currentClassInfo.front()) at the globals level.

**struct** \_frama\_c\_rtti\_name\_info\_content XXX::\_frama\_c\_base\_classes[xxx] = { {…, …, …}, …, {…, …, …} };

where XXX stands for makeQualifiedName(\_currentClass.front()) and xxx stands for \_currentClassInfo.front().\_virtualDerivationsPosition.size() + \_currentClassInfo.front().\_derivationsPosition.size(); this definition is compatible with the declaration done by the method insertStaticBaseClassesDeclaration.

If \_currentClassInfo.front().\_virtualDerivationsPosition.empty() = **false**, our method also generates the following declaration for classes deriving from \_currentClassInfo.front():

**struct** \_frama\_c\_rtti\_name\_info\_content XXX\_frama\_c\_bare::\_frama\_c\_base\_classes[xxx] = { {…, …, …}, … };

For each 0 ≤ i < \_currentClassInfo.front().\_derivationsPosition.size(), derivationIter of type InheritancePosition:: iterator points onto the ith static base class.

* \_frama\_c\_base\_classes[i].value = & makeQualifiedName(derivationIter->first)::\_frama\_c\_rtti\_name\_info;
* \_frama\_c\_base\_classes[i].shift\_object = (void\*) 0 - (void\*) ((derivationIter->first\*) (\_currentClass.front()\*) 0);
* \_frama\_c\_base\_classes[i].shift\_vmt = derivationIter->second;

This last generation only occurs for classes having virtual base classes:

For each \_currentClassInfo.front().\_derivationsPosition.size() ≤ i <

\_currentClassInfo.front().\_virtualDerivationsPosition.size() + \_currentClassInfo.front().\_derivationsPosition.size(),

virtualDerivationIter of type VirtualInheritancePosition::iterator points onto the (virtualIndex = i - …)th virtual base class.

* \_frama\_c\_base\_classes[i].value =

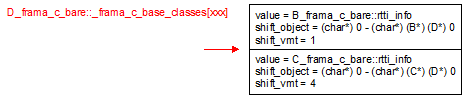
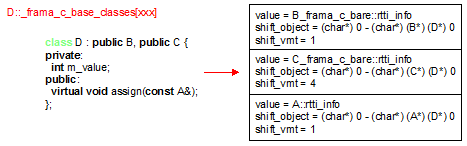
& makeQualifiedName(\_currentClassInfo.front()\_virtualBaseClassTable[virtualIndex])::\_frama\_c\_rtti\_name\_info;

* \_frama\_c\_base\_classes[i].shift\_object =

(void\*) 0 - (void\*) ((\_currentClassInfo.front()\_virtualBaseClassTable[virtualIndex]\*) (\_currentClass.front()\*) 0);

* \_frama\_c\_base\_classes[i].shift\_vmt = \*virtualDerivationIter;

For the example 3,



Our method is called by the method exitClass, between the methods insertStaticVMTDefinition, insertDeclStaticVMTHeaderDefinition, insertStaticRTTIDefinition and insertStaticVMTHeaderDefinition for each class that has virtual methods.

* + The method addBareToQualification,
  + the method insertStaticBaseClassesDeclaration,
  + the classes ClassInfo, ClassInfo:: InheritancePosition,
  + the method exitClass,
  + the methods insertStaticBaseClassesDefinition, insertDeclStaticVMTHeaderDefinition, insertStaticRTTIDefinition and insertStaticVMTHeaderDefinition.

**void** insertStaticRTTIDeclaration(ForwardReferenceList& classContent, location classLoc);

Insert the declaration

**struct** \_frama\_c\_rtti\_name\_info\_node \_frama\_c\_rtti\_name\_info;

to declare the run-time type information of the current class (see the field \_currentClassInfo.front()).

Its definition is defined by the method insertStaticRTTIDefinition. Our method is called by the method exitClass, between the methods insertStaticVMTDeclaration, insertStaticBaseClassesDeclaration, and the method insertStaticVMTHeaderDeclaration for each class that has virtual methods.

* + The method insertStaticRTTIDefinition,
  + the method exitClass,
  + the methods insertStaticVMTDeclaration, insertStaticBaseClassesDeclaration, insertStaticVMTHeaderDeclaration.

qualified\_name insertStaticRTTIDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc, qualified\_name vmtName, qualified\_name nameBaseClasses);

Insert the definition of the run-time type information (or rtti for short) of the current class (see the field \_currentClassInfo.front()) at the globals level.

**struct** XXX::\_frama\_c\_rtti\_name\_info\_node \_frama\_c\_rtti\_name\_info = { … };

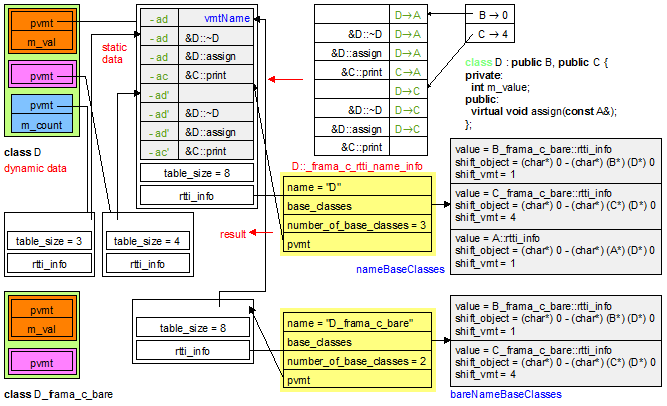
where XXX stands for makeQualifiedName(\_currentClass.front()); this definition is compatible with the declaration done by the method insertStaticRTTIDeclaration. This rtti structure is built above vmtName that is the virtual method table and above nameBaseClasses that is the access to the base classes of our class.

If \_currentClassInfo.front().\_virtualDerivationsPosition.empty() = **false**, our method also generates the following declaration for classes deriving from \_currentClassInfo.front():

**struct** XXX\_frama\_c\_bare::\_frama\_c\_rtti\_name\_info\_node \_frama\_c\_rtti\_name\_info = { … };

* + vmtName is the result of the call to the method insertStaticVMTDefinition.
  + nameBaseClasses is the result of the call to the method insertStaticBaseClassesDefinition.

For the example 3, our method generates the part yellow-underlined on the following schema and it returns result.



* + The method insertStaticRTTIDeclaration,
  + the method exitClass,
  + the methods insertStaticVMTDefinition, insertStaticBaseClassesDefinition, insertDeclStaticVMTHeaderDefinition and insertStaticVMTHeaderDefinition.

**void** insertStaticVMTHeaderDeclaration(ForwardReferenceList& classContent, location classLoc);

Insert the declaration

**struct** \_frama\_c\_vmt \_frama\_c\_vmt\_header\_for\_shift\_nnn\_0;

…

**struct** \_frama\_c\_vmt \_frama\_c\_vmt\_header\_for\_shift\_nnn\_(x-1);

**struct** \_frama\_c\_vmt \_frama\_c\_vmt\_header;

to declare the root object data structure (virtual method table and run-time type information) on which point the objects created by the program for the current class (see the field \_currentClassInfo.front()).

x stands for the number of pvmt (pointers on virtual method table) that are inside an object of type \_currentClassInfo.front().

nnn\_i where 0 ≤ i < x, is the shift in the virtual method table of \_currentClassInfo.front() of the ith pointer inside the object on the virtual method table.

Be careful that nnn\_i should be unique.

The definitions of \_frama\_c\_vmt\_header, \_frama\_c\_vmt\_header\_for\_shift\_nnn\_i are defined by the method insertStaticVMTHeaderDefinition. The method insertDeclStaticVMTHeaderDefinition only declares it at global level for the method insertStaticRTTIDefinition that generates a backward link to the root data structure. Our method is called by the method exitClass, after the methods insertStaticVMTDeclaration, insertStaticBaseClassesDeclaration and insertStaticRTTIDeclaration for each class that has virtual methods.

* + The methods insertStaticVMTHeaderDefinition and insertDeclStaticVMTHeaderDefinition,
  + the method exitClass,
  + the methods insertStaticVMTDeclaration, insertStaticBaseClassesDeclaration, insertStaticRTTIDeclaration.

**void** insertDeclStaticVMTHeaderDefinition(ForwardReferenceList& globals, location classLoc, qualified\_name vmtName);

Insert at globals level the declaration of the main root object data structure (virtual method table and run-time type information) on which point the objects created by the program for the current class (see the field \_currentClassInfo.front()).

**struct** XXX::\_frama\_c\_vmt \_frama\_c\_vmt\_header;

where XXX stands for makeQualifiedName(\_currentClass.front()); It is a declaration for the method insertStaticRTTIDefinition that generates a backward link to this root data structure. This definition is compatible with the declaration done by the method insertStaticVMTHeaderDeclaration. This root data structure uses vmtName as a similar (template) global implementation name.

Our method also insert at globals level

**struct** XXX\_frama\_c\_bare::\_frama\_c\_vmt \_frama\_c\_vmt\_header;

if \_currentClassInfo.front() has virtual base classes.

1. vmtName is the result of the call to the method insertStaticVMTDefinition.
   * The method addBareToQualification,
   * the methods insertStaticVMTHeaderDefinition and insertStaticVMTHeaderDeclaration,
   * the method exitClass,
   * the methods insertStaticVMTDefinition, insertStaticBaseClassesDefinition, insertStaticRTTIDefinition.

**void** insertStaticVMTHeaderDefinition(**const** Clang\_utils& utils, ForwardReferenceList& globals, location classLoc, qualified\_name vmtName, qualified\_name nameRtti);

Insert at globals level the declaration of the root object data structures (virtual method table and run-time type information) on which point the objects created by the program for the current class (see the field \_currentClassInfo.front()).

**struct** \_frama\_c\_vmt XXX::\_frama\_c\_vmt\_header = { …, …, … };

**struct** \_frama\_c\_vmt XXX::\_frama\_c\_vmt\_header\_for\_shift\_nnn\_0 = { …, …, … };

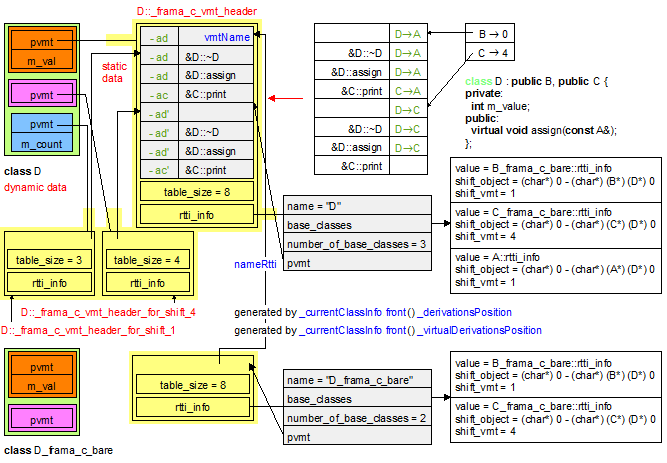
…

**struct** \_frama\_c\_vmt XXX::\_frama\_c\_vmt\_header\_for\_shift\_nnn\_(x-1) = { …, …, … };

where XXX stands for makeQualifiedName(\_currentClass.front()); This definition is compatible with the declaration done by the method insertStaticVMTHeaderDeclaration. This root data structures are built above vmtName that is the virtual method table and above nameRtti that is the run-time type information of the current class.

* + vmtName is the result of the call to the method insertStaticVMTDefinition,
  + nameRtti is the result of the call to the method insertStaticRTTIDefinition.

For the example 3, our method generates the part yellow-underlined on the following schema.



* + The method insertStaticVMTHeaderDeclaration,
  + the method exitClass,
  + the methods insertDeclStaticVMTHeaderDefinition, insertStaticVMTDefinition, insertStaticBaseClassesDefinition, insertStaticRTTIDefinition.

**bool** retrieveStaticInheritancePathBetween(**const** clang::CXXRecordDecl\* derived, **const** clang::CXXRecordDecl\* base, InheritancePath& result, **const** Clang\_utils& utils, VirtualInheritancePath\* virtualResult = **nullptr**) **const**;

Set in result (and optionally in virtualResult) the hierarchic class path from derived to base and return **true** if and only if such a path exists. If this method is called with virtualResult = **nullptr**, that means that the search does not consider virtual base classes. The algorithm simply uses a depth first search starting from the base classes of derived until it reaches base. If base is not reached, our method returns **false**.

Our method is called in both contexts by the public method retrieveInheritancePathBetween for its depth first search algorithm.

1. There is at most one path from derived to base – the clang compilation guaranties it.
   * The method getBasePosition and the classes InheritancePath, VirtualInheritancePath,
   * the method retrieveInheritancePathBetween.

**void** addBareToQualification(qualified\_name& name) **const**;

Add the suffix “\_frama\_c\_bare” to the name of the last record context in the qualification of name. Our method enables to simply create data structures to handle with virtual inheritance. These data structures are a copy of the original data structure but without the fields dedicated to the virtual base classes.

Our method is a utility method called by the following public methods: insertStaticBaseClassesDefinition, insertDeclStaticVMTHeaderDefinition, insertStaticRTTIDefinition, insertStaticVMTHeaderDefinition.

* + name.prequalification ≠ **nullptr**,
  + name.prequalification ends with a structure (\_qualification::tag\_qualification = QSTRUCTORCLASS or \_qualification::tag\_qualification = QTEMPLATEINSTANCE but not QNAMESPACE)
  + The class denoted by name.prequalification has virtual base classes.

1. The methods insertStaticBaseClassesDefinition, insertDeclStaticVMTHeaderDefinition, insertStaticRTTIDefinition, insertStaticVMTHeaderDefinition.

Public methods

**void** insertVMTAndRttiPrelude(ForwardReferenceList& globals, location loc);

Generate the following definitions at the beginning of each translation unit:

*/\* generated by insertVMTContentPrelude \*/*

**struct** \_frama\_c\_vmt\_content {

**void** (\*method\_ptr)();

**int** shift\_this;

};

**struct** \_frama\_c\_rtti\_name\_info\_node;

*/\* generated by insertVMTTypePrelude \*/*

**struct** \_frama\_c\_vmt {

**struct** \_frama\_c\_vmt\_content\* table;

**int** table\_size;

**struct** \_frama\_c\_rtti\_name\_info\_node\* rtti\_info;

};

*/\* generated by insertRTTIInfoPrelude \*/*

**struct** \_frama\_c\_rtti\_name\_info\_content {

**struct** \_frama\_c\_rtti\_name\_info\_node\* value;

**int** shift\_object;

**int** shift\_vmt;

};

**struct** \_frama\_c\_rtti\_name\_info\_node {

**const** **char**\* name;

**struct** \_frama\_c\_rtti\_name\_info\_content\* base\_classes;

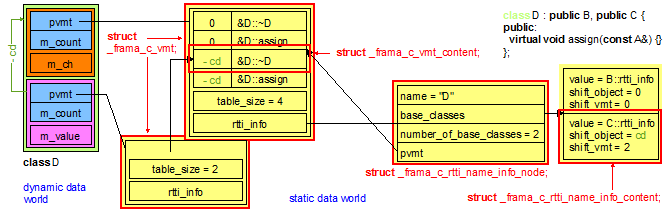
**int** number\_of\_base\_classes;

**struct** \_frama\_c\_vmt\* pvmt;

};

It defines the data structure for building the virtual method table and the run-time type information (or rtti for short).

The following schema illustrates these data structures on an example of multiple inheritance (see example 1)



The implementation calls the methods insertVMTContentPrelude, insertVMTTypePrelude, insertRTTIInfoPrelude.

It is called by the constructor of the class Visitor as the first code generation.

* + The methods insertVMTContentPrelude, insertVMTTypePrelude, insertRTTIInfoPrelude,
  + the constructor Visitor::Visitor(FILE\*, clang::CompilerInstance&).

**void** exitClass(**const** Clang\_utils& utils, ForwardReferenceList& content, ForwardReferenceList& globals, location classLoc);

Close the virtual method table and the run-time type information of the current class \_currentClass.front(). This method is called when all the virtual methods of the class are known, at the end of the declaration of the current class. By symmetry the method enterClass had been called at the beginning of the declaration of the current class.

If our class has virtual method or virtual base classes, our method calls the following methods:

* insertStaticVMTDeclaration to declare the static data structure \_frama\_c\_vmt\_content inside the class,
* insertStaticBaseClassesDeclaration to declare the static data structure \_frama\_c\_rtti\_name\_info\_content inside the class,
* insertStaticRTTIDeclaration to declare the static data structure \_frama\_c\_rtti\_name\_info\_node inside the class,
* insertStaticVMTHeaderDeclaration to declare the static data structure \_frama\_c\_vmt\_header inside the class, and depending on the base classes the \_frama\_c\_vmt\_header\_for\_shift\_... data structures,
* addPvmtSetter to set the right virtual method table to the constructors and the destructors,
* insertRttiPrelude at global level to declare all the structures required by the static data structures of the current class,
* insertStaticVMTDefinition to define the static data structure \_frama\_c\_vmt\_content outside the class,
* insertStaticBaseClassesDefinition to define the static data structure \_frama\_c\_rtti\_name\_info\_content outside the class,
* insertDeclStaticVMTHeaderDefinition to declare the static data structure \_frama\_c\_vmt\_header outside the class,
* insertStaticRTTIDefinition to define the static data structure \_frama\_c\_rtti\_name\_info\_node outside the class,
* insertStaticVMTHeaderDeclaration to define the static data structure \_frama\_c\_vmt\_header outside the class, and depending on the base classes the \_frama\_c\_vmt\_header\_for\_shift\_... data structures,
* DelayedMethodCalls::updateWith to complete the calls to virtual method (and access to base classes) of our class by the knowledge of the virtual method table.
* ClassInfo::swap to transfer the content of \_currentClassInfo.front() into \_classInfoTable,
* and exitClass() at the end to pop the class from the stack.

Our method is called by the method Visitor::postLexicalVisitRecordDecl at the end of each class definition.

1. The method enterClass should have been called with the class we are exiting.
2. * The method enterClass,

* the methods nsertStaticVMTDeclaration, insertStaticBaseClassesDeclaration, insertStaticRTTIDeclaration, insertStaticVMTHeaderDeclaration,
* the method addPvmtSetter,
* the method insertRttiPrelude,
* the methods insertStaticVMTDefinition, insertStaticBaseClassesDefinition, insertDeclStaticVMTHeaderDefinition, insertStaticRTTIDefinition, insertStaticVMTHeaderDeclaration,
* the methods DelayedMethodCalls::updateWith and exitClass(),
* the method Visitor::postLexicalVisitRecordDecl.

**const** ClassInfo\* getClassInfo(**const** clang::CXXRecordDecl\* source) **const**;

Return the ClassInfo associated to the class source. This information is complete if source is present in \_classInfoTable. It is under construction if it is in the fields \_currentClass / \_currentClassInfo. Under construction means that the base classes are present, but some virtual methods may be missing. Our method returns **nullptr** if source is neither present in \_classInfoTable nor in \_currentClass.

Our method is a helper method for the method getPvmt as example.

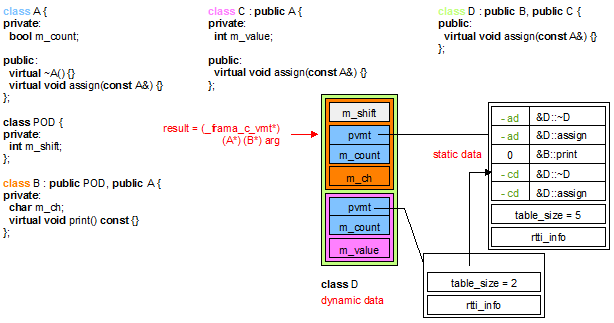
* + The fields \_classInfoTable, \_currentClass and \_currentClassInfo,
  + the method getPvmt.

expression getPvmt(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* source, expression arg) **const**;

Return an expression that gives an access to the virtual method table of arg; the type of arg is a (**this**) pointer on source.

The implementation calls the methods getClassInfo, ClassInfo::hasPvmtAsFirstField, ClassInfo::getPvmtField if the pointer on the virtual method table is not the first field of the class source.

The following schema illustrates the result of our method on the example 2:



* + The field \_classInfoTable and the methods ClassInfo::hasPvmtAsFirstField, ClassInfo::getPvmtField,
  + the method getClassInfo,
  + the methods Visitor::makeCastExpression, Visitor::makeDynamicCastExpression.

**void** addPvmtSetter(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* source, */\* statement \*/* list& insertionPoint, location loc);

Assign the pointers of the virtual method table of source at the insertion point insertionPoint.

Depending on the result of \_currentClassInfo.front()->hasPvmtAsFirstField(), our method inserts

\*((\_frama\_c\_vmt\*\*) this) = &...::\_frama\_c\_vmt\_header;

if \_currentClassInfo.front()->hasPvmtAsFirstField() = **false**, or it inserts

\*((\_frama\_c\_vmt\*\*) &this->pvmt) = &...::\_frama\_c\_vmt\_header;

if \_currentClassInfo.front()->hasPvmtAsFirstField() = **true** with this->pvmt defined by \_currentClassInfo.front()‑>getPvmtField().

It also sets the intermediate pmvt for the base classes with shift and virtual methods and also for the virtual base classes.

Our method is called in two contexts:

* if the visited code is not attached to the current class, our method is directly called to set the right virtual method table in the destructors and the constructors of methods,
* if the visited code is attached to the current class (whose method table is not known), our method is called in exitClass on each \_pvmtInsertionPoints after the visited code had called addPvmtSetter(*/\* statement \*/* list&) to temporarily store its argument in the field \_pvmtInsertionPoints.
  + The methods ClassInfo::hasPvmtAsFirstField, ClassInfo::getPvmtField,
  + the methods exitClass, insertDeclStaticVMTHeaderDefinition,
  + the field \_pmvtInsertionPoints and the method addPvmtSetter(*/\* statement \*/* list&),
  + the methods Visitor::insertConstructorPreambleIn, Visitor::insertDestructorPreambleIn, Visitor::VisitFunctionDecl..

**int** addDerivation(**const** clang::CXXRecordDecl\* source, **bool** isVirtual);

Add source as a base class of \_currentClassInfo.front(). isVirtual indicates if the base class is a virtual base class or a standard base class. It returns the position in the virtual method table where newly declared virtual methods should be registered.

The implementation looks for the ClassInfo attached to source. If it finds one, it simply calls the method ClassInfo::addDerivation. A more detailed description can be found at this method.

This method is called by Visitor::makeInheritanceList for each base class of a given class.

* + The methods VirtualMethodInfo::addInherits, VirtualMethodInfo::setVirtualInherits, ClassInfo::addDerivation,
  + the method Visitor::makeInheritanceList.

**int** getBasePosition(**const** clang::CXXRecordDecl\* derived, **const** clang::CXXRecordDecl\* base, **bool**& isVirtual) **const**;

Return the index in the virtual method table (see the field ClassInfo::\_virtualMethodTable) where the own virtual methods of base are defined. The implementation calls the method ClassInfo::getBasePosition on the result of getClassInfo(derived).

A more detailed description can be found at the level of ClassInfo::getBasePosition.

This method is called by the method retrieveStaticInheritancePathBetween in case of virtual inheritance and by the method retrieveBaseIndex that provides an interface for the methods Visitor::makeDerivedToBasePointerCastExpression, Visitor:: makeDerivedToBaseReferenceCastExpression and Visitor:: insertConstructorPreambleIn.

1. base is a direct base class of our class or it is a virtual base class (see the method ClassInfo::isVirtualBase)
   * The field ClassInfo::\_virtualMethodTable and the method ClassInfo::getBasePosition,
   * the method getClassInfo,
   * the methods retrieveStaticInheritancePathBetween, retrieveInheritancePathBetween, retrieveBaseIndex,
   * the methods Visitor::makeDerivedToBasePointerCastExpression, Visitor:: makeDerivedToBaseReferenceCastExpression, Visitor::makeBaseToDerivedPointerCastExpression and Visitor:: insertConstructorPreambleIn.

**void** retrieveMethodIndex(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* classCaller, clang::CXXMethodDecl\* methodCalled, int64\_t\* methodIndex, */\* inheritance \*/* list\* shiftObject, */\* inheritance \*/* list\* shiftPvmt);

Return in methodIndex the index of method in the virtual method table (see the field ClassInfo::\_virtualMethodTable) of classCaller. At the end shiftObject contains the path that enables to define the shift of the object from the actual shift parameter to the formal parameter of the called method. shiftPvmt contains the path that enables to access to the pointer of the virtual method table inside the object.

Note that the virtual method table of classCaller may be under construction and so incomplete. In such a case, our method only registers the address of methodIndex, shiftObject and shiftPvmt (see the field \_delayedMethodCalls, the class DelayedMethodCalls and the method DelayedMethodCalls::addMethodCall). When the complete virtual method table is available (see the method exitClass) \*methodIndex, \*shiftObject and \*shiftPvmt receive their final value (see the method DelayedMethodCalls::updateWith).

In the standard case, \*methodIndex receives the result of ClassInfo::getIndexOfMethod with the ClassInfo found in \_classInfoTable. The couple InheritancePath / VirtualInheritancePath returned by the method ClassInfo::getIndexOfMethod enables to build \*shiftObject. If the ClassInfo found in \_classInfoTable has a ClassInfo::hasPvmtAsFirstField() then ClassInfo::getPvmtField() enables to build \*shiftPvmt.

Our method is called by the method Visitor::makeMemberCallExpression to translate every call to a virtual method into the intermediate representation.

* + The fields \_classInfoTable, \_delayedMethodCalls and ClassInfo::\_virtualMethodTable,
  + the methods ClassInfo::getIndexOfMethod, ClassInfo::hasPvmtAsFirstField, ClassInfo::getPvmtField,
  + the class DelayedMethodCalls and the methods DelayedMethodCalls::addMethodCall, DelayedMethodCalls::updateWith, exitClass,
  + the method retrieveBaseIndex, retrieveInheritancePathBetween,
  + the method Visitor::makeMemberCallExpression.

**void** retrieveBaseIndex(**const** Clang\_utils& utils, **const** clang::CXXRecordDecl\* classCaller, clang::CXXRecordDecl\* baseClass, int64\_t\* accessIndex);

Return in \*accessIndex the index in the virtual method table (see the field ClassInfo::\_virtualMethodTable) of classCaller where the own virtual methods of baseClass are defined.

The implementation simply looks at the index associated to base in the fields ClassInfo::\_virtualDerivationsPosition / ClassInfo::\_derivationsPosition.

Note that the virtual method table of classCaller may be under construction and so incomplete. In such a case, our method only registers the address of accessIndex (see the field \_delayedMethodCalls, the class DelayedMethodCalls and the method DelayedMethodCalls::addFieldAccess). When the complete virtual method table is available (see the method exitClass) \*accessIndex receives its final value (see the method DelayedMethodCalls::updateWith).

In the standard case, \*accessIndex receives the result of ClassInfo::getBasePosition with the ClassInfo found in \_classInfoTable.

Our method is called by the method Visitor::makeDerivedToBasePointerCastExpression, Visitor::makeDerivedToBaseReferenceCastExpression, Visitor:: insertConstructorPreambleIn to translate every (unknown) access to a virtual base class into the intermediate representation.

1. baseClass is a direct base class of classCaller or it is a virtual base class.
   * The fields \_classInfoTable, \_delayedMethodCalls, ClassInfo::\_virtualMethodTable, ClassInfo::\_virtualDerivationsPosition, ClassInfo::\_derivationsPosition,
   * the methods ClassInfo::getBasePosition, ClassInfo::isVirtualBase,
   * the methods getBasePosition, retrieveMethodIndex, retrieveInheritancePathBetween,
   * the class DelayedMethodCalls and the methods DelayedMethodCalls::addFieldAccess, DelayedMethodCalls::updateWith, exitClass,
   * the method Visitor::makeDerivedToBasePointerCastExpression, Visitor::makeDerivedToBaseReferenceCastExpression, Visitor:: insertConstructorPreambleIn.

**void** retrieveInheritancePathBetween(**const** clang::CXXRecordDecl\* derived, **const** clang::CXXRecordDecl\* base, InheritancePath& result, VirtualInheritancePath& virtualResult, **const** Clang\_utils& utils) **const**;

Set in result (and optionally in virtualResult) the hierarchic class path from derived to base.

The implementation calls the method retrieveStaticInheritancePathBetween for its depth first search algorithm on the class hierarchy issued from derived. If derived has known virtual base classes, the implementation calls retrieveStaticInheritancePathBetween from each virtual base class until it returns **true**. If no such call returns **true** or if derived has no virtual base class then a call to retrieveStaticInheritancePathBetween that authorizes the inspection of virtual base classes will return the result, that is the hierarchic class path from derived to base.

Our method is called by the methods Clang\_utils::retrieveTypeOfField (field on a base class), Visitor::makeBaseToDerivedPointerCastExpression (virtualResult.first should remain unchanged to **nullptr**), Visitor::makeBaseToDerivedReferenceCastExpression (virtualResult.first should remain unchanged to **nullptr**), Visitor::makeDerivedToBasePointerCastExpression, Visitor::makeDerivedToBaseReferenceCastExpression, Visitor::insertConstructorPreambleIn, but also by the method insertStaticVMTDefinition for its depth first search algorithm.

1. There is one and only one path from derived to base – the clang compilation guaranties it.
   * The method retrieveStaticInheritancePathBetween and the classes InheritancePath, VirtualInheritancePath,
   * the methods Clang\_utils::retrieveTypeOfField, Visitor::makeBaseToDerivedPointerCastExpression, Visitor::makeBaseToDerivedReferenceCastExpression, Visitor::makeDerivedToBasePointerCastExpression, Visitor::makeDerivedToBaseReferenceCastExpression, Visitor::insertConstructorPreambleIn and insertStaticVMTDefinition.

The VisitTable Unit

The main class of this library is the class VisitTable. It stores all the clang::Decl that have been visited. It stores information on the template instances waiting for declarations but also on the waited declarations. As soon as a waited declaration is encountered, VisitTable automatically generates the code of the template instances.

The generation is complex. For example, for a template class instance, we should generate:

* the names of classes issued from the instance parameters,
* the name of the class instance,
* the declaration of the classes issued from the instance parameters,
* the data structure declaration of the class instance,
* the declaration of the classes used by the methods of the class instance,
* the methods of the class instance.

Our representative example is the following code:

|  |  |  |
| --- | --- | --- |
| **template** <**class** A, **class** B, **typename** T>  **class** X : **public** A {  **private**:  **typename** B::C \_field;  T\* \_pointer;  **public**:  X(**const** B& source)  : \_field(source), \_pointer(**nullptr**) {}  ~X() { **if** (\_pointer) **delete** \_pointer; }  **void** setPointer(T\* pointer)  { **if** (\_pointer && \_pointer != pointer)  **delete** \_pointer;  \_pointer = pointer;  }  }; | **class** Foo {  **public**:  **int** \_value;  };  **class** Bar {  **public**:  **typedef** **int** C;  **int** \_value;  };  **class** Bar2 {  **public**:  **int** \_content;  }; | **int** main() {  X<Foo, Bar, Bar2> x;  **return** 0;  } |

Figure 3: simple example of template code

For this example, here is the Cabs code to generate:

|  |  |  |  |
| --- | --- | --- | --- |
| **class** Foo;  **class** Bar;  **class** Bar2;  **class** X<Foo, Bar, Bar2>;  **class** Foo {  **public**:  **int** \_value;  };  **class** Bar {  **public**:  **typedef** **int** C;  **int** \_value;  }; | **class** X<Foo, Bar, Bar2>  : **public** Foo {  **private**:  Bar::C \_field;  Bar2\* \_pointer;  **public**:  X(**const** Bar& source)  : \_field(source),  \_pointer(**nullptr**) {}  ~X();  **void** setPointer  (Bar2\* pointer);  }; | **class** Bar2 {  **public**:  **int** \_content;  };  X<Foo, Bar, Bar2>::~X()  { **if** (\_pointer)  **delete** \_pointer;  } | **void**  X<Foo, Bar, Bar2>  ::setPointer(Bar2\* pointer)  { **if** (\_pointer  && \_pointer != pointer)  **delete** \_pointer;  \_pointer = pointer;  }  **int** main() {  X<Foo, Bar, Bar2> x;  **return** 0;  } |

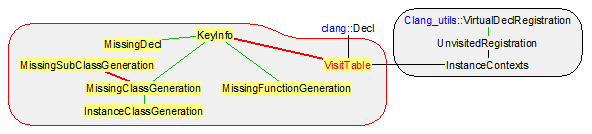
Figure 4: generated Cabs for template code instances

Our unit contains the information related to the declarations whose visitation has an impact on the template instance generation. Hence all visited declarations (class, function, typedef, constant) should be registered to know if an instance can have access to its definition of if it has to wait for it.

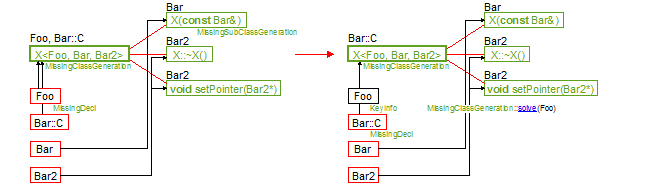
The main class of this unit is the class VisitTable that is a map from clang::Decl to visit information on the declaration. 4 types of information KeyInfo are available:

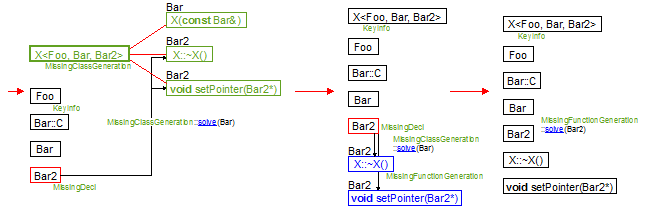
* The name of a declaration has not been encountered. It is represented by the absence of entry in the VisitTable map.
* The name of a declaration has been encountered but not its body. It is represented by a connection clang::Decl → MissingDecl in the VisitTable map.
* The declaration has been visited but cannot be generated due to missing declarations. It is represented by a connection clang::Decl → MissingFunctionGeneration or clang::Decl → MissingClassGeneration in the VisitTable map. During a class instance visitation we do not know if the generation of the translation\_unit\_decl or class\_decl will be effective in Cabs at the end of the visit. So we create an InstanceClassGeneration deriving from MissingClassGeneration that is likely to produce template instances in cascade with its field std::vector<KeyInfo\*> InstanceClassGeneration::\_waitingDecls if the generation does not depend of missing classes. The InstanceClassGeneration is then translated into a simple KeyInfo in the table. In the alternate case, it is translated into a MissingClassGeneration since the field InstanceClassGeneration::\_waitingDecls has been moved in the MissingDecl::\_waitingDecls of the instance parameters. The reason is that they can directly trigger the generation when the instance parameters are generated.
* The declaration has been visited and has been generated. It is represented by a connection clang::Decl → KeyInfo in the VisitTable map.

The inheritance graph of this unit is defined on the following schema.



The following sequence of schemas describes the evolution of the VisitTable during the visit of clang declarations in Figure 3: simple example of template code. At the end of the algorithm VisitTable::isComplete returns **true**, which means that all clang declarations have produced their Cabs corresponding.



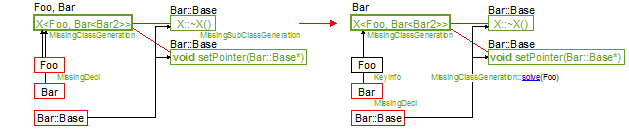


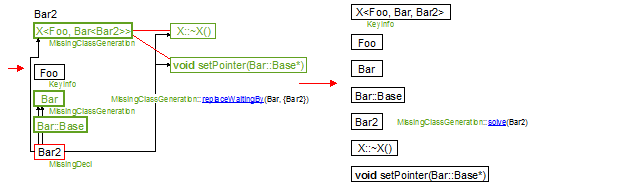
To illustrate another point of the generation algorithm, let us introduce the following example that causes partial instance and the call to KeyInfo::replaceWaitingBy.

|  |  |  |
| --- | --- | --- |
| **template** <**class** A, **class** B>  **class** X : **public** A {  **private**:  **typename** B::C \_field;  **typename** B::Base\* \_pointer;  **public**:  X(**const** B& source)  : \_field(source), \_pointer(**nullptr**) {}  ~X() { **if** (\_pointer) **delete** \_pointer; }  **void** setPointer(**typename** B::Base \* pointer)  { **if** (\_pointer && \_pointer != pointer)  **delete** \_pointer;  \_pointer = pointer;  }  }; | **class** Foo {  **public**:  **int** \_value;  };  **template**<**class** T>  **class** Bar {  **public**:  **typedef** T Base;  **typedef** T C;  T\_value;  };  **class** Bar2 {  **public**:  **int** \_content;  }; | **int** main() {  X<Foo, Bar<Bar2> > x;  **return** 0;  } |

Figure 5: variation for the Figure 3 example

On this example, the next figure describes the evolution of the VisitTable during the visit of clang declarations. At the end of the algorithm VisitTable::isComplete also returns **true**: all clang declarations have produced their Cabs corresponding.





The class KeyInfo

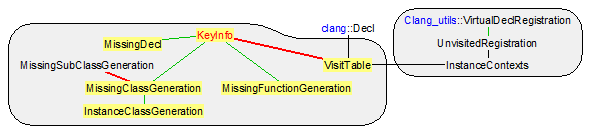
The class KeyInfo is a virtual base class summarizing the visit info available for a clang::Decl. As it is preferable to keep the key available from the KeyInfo, we use the container std::set<KeyInfo\*> to register the information in the table.

A KeyInfo entry represents an encountered name. If just the name is encountered, then the KeyInfo should be a MissingDecl.

If the declaration is visited and if the generation has occurred, then the entry is actually a KeyInfo.

If the declaration is visited and if some declarations are missing for its generation then the entry is either a MissingClassGeneration or a MissingFunctionGeneration.

The inheritance graph of this class is defined as following.



On the “Figure 3: simple example of template code”, the visit of class Foo creates a pure KeyInfo whereas the visit of X creates a MissingClassGeneration that is automatically translated into a pure KeyInfo after the visit of Bar.

Fields of the class KeyInfo

**const** clang::Decl\* \_key;

This field represents the clang declaration that has been visited or the clang declaration we are waiting for its visit. This key is used to sort the KeyInfo within the class VisitTable. The properties we are looking for is the uniqueness of the key in the table and quick search function. That is why a sort based on pointer is sufficient even it is non-deterministic across different compilations. This key is not **nullptr**.

Declaration of the class KeyInfo

**class** KeyInfo {

**private**:

**const** clang::Decl\* \_key;

**friend** **class** VisitTable;

**public**:

KeyInfo(**const** clang::Decl\* key) : \_key(key) {}

KeyInfo(**const** KeyInfo& source) : \_key(source.\_key) {}

**virtual** ~KeyInfo() {}

**virtual** **bool** isMissingDecl() **const** { **return** **false**; }

**virtual** **bool** isGenerationMissing() **const** { **return** **false**; }

**virtual** **bool** isClassGenerationMissing() **const** { **return** **false**; }

**virtual** **bool** isInstanceClass() **const** { **return** **false**; }

**virtual** **bool** isFunctionGenerationMissing() **const** { **return** **false**; }

**virtual** **void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls) { assert(**false**); }

**virtual** **bool** solve(**const** clang::Decl\* decl, ForwardReferenceList& globals, VisitTable& table) { assert(**false**); }

**virtual** **bool** isComplete() **const** { **return** **true**; }

**const** clang::Decl\* key() **const** { **return** \_key; }

**class** Less {

**public**:

**bool** **operator**()(**const** KeyInfo\* first, **const** KeyInfo\* second) **const** { **return** first->\_key < second->\_key; }

};

};

Methods of the class KeyInfo

Public methods

**virtual** **bool** isMissingDecl() **const**;

Returns **true** if and only if our KeyInfo is a MissingDecl. This means that the name of \_key a declaration has been encountered but not its body. The method is used in this case, to know if the declaration is available (see the method VisitTable::hasVisited).

1. If the method returns **true**, our KeyInfo supports the type MissingDecl.
   * The methods isGenerationMissing, isClassGenerationMissing, isFunctionGenerationMissing,
   * the method isComplete,
   * the method VisitTable::hasVisited,
   * the methods VisitTable::setInstanceClassAsComplete, VisitTable::addWaitFor, VisitTable::addDeclaration, VisitTable::addInstanceClass, VisitTable::addIncompleteClass, VisitTable::addIncompleteFunction.

**virtual** **bool** isGenerationMissing() **const**;

Returns **true** if and only if our KeyInfo is a MissingFunctionGeneration or a MissingClassGeneration. This means that the declaration has been visited but cannot be generated due to missing declarations.

1. If the method returns **true**, you should call isFunctionGenerationMissing or isClassGenerationMissing to know if our KeyInfo supports the type MissingFunctionGeneration or a MissingClassGeneration.
   * The methods isMissingDecl, isClassGenerationMissing, isFunctionGenerationMissing,
   * the method isComplete,
   * the method VisitTable::hasVisited.

**virtual** **bool** isClassGenerationMissing() **const**;

Returns **true** if and only if our KeyInfo is a MissingClassGeneration. This means that the class declaration \_key of type clang::RecordDecl has been visited but cannot be generated due to missing declarations.

1. If the method returns **true**, our KeyInfo supports the type MissingClassGeneration.
   * The methods isMissingDecl, isGenerationMissing, isFunctionGenerationMissing,
   * the method isComplete,
   * the method VisitTable::hasVisited.

**virtual** **bool** isInstanceClass() **const**;

Returns **true** if and only if our KeyInfo is an InstanceClassGeneration. This means that the class declaration \_key of type clang::RecordDecl is currently visited. For the moment, we do not know if the class declaration could be generated or not at the end of the visit. In the case its inherited field \_additionalWaitDeclarations remains empty, the generation will occur and our information entry is translated into a pure KeyInfo. In the alternate case, the generation is delayed until the visit of the clang::Decl and at the end of the visit our entry is translated in a pure MissingClassGeneration.

1. If the method returns **true**, our KeyInfo supports the type InstanceClassGeneration.
   * The methods isClassGenerationMissing, isMissingDecl, isGenerationMissing, isFunctionGenerationMissing,
   * the method isComplete,
   * the method VisitTable::hasVisited,
   * the methods VisitTable::setInstanceClassAsComplete, Visitor::postVisitRecordDecl.

**virtual** **bool** isFunctionGenerationMissing() **const**;

Returns **true** if and only if our KeyInfo is a MissingFunctionGeneration. This means that the class declaration \_key of type clang::FunctionDecl has been visited but cannot be generated due to missing declarations.

1. If the method returns **true**, our KeyInfo supports the type MissingFunctionGeneration.
   * The methods isMissingDecl, isGenerationMissing, isClassGenerationMissing,
   * the method isComplete,
   * the method VisitTable::hasVisited.

**virtual** **bool** isComplete() **const**;

Returns **true** if and only if our KeyInfo has been generated. So this method returns **true** for pure KeyInfo.

The method is called by VisitTable::isComplete to verify that at the end of a translation unit visit all declarations have been generated and in particular all the template instances generated by clang.

* + The methods isMissingDecl, isGenerationMissing, isFunctionGenerationMissing, isClassGenerationMissing,
  + the method VisitTable::isComplete and the method Visitor::HandleTranslationUnit.

**virtual** **void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls);

This method is called on incomplete entries (see the method isGenerationMissing) to replace the dependence to oldDecl with the new dependence newDecls. The main concerned fields are MissingFunctionGeneration::\_waitDeclarations and MissingClassGeneration::\_waitDeclarations and they should not contain multiple references to the same clang::Decl.

The method is called when oldDecl is visited although it was waited by other (isGenerationMissing()) KeyInfo and when the generation of oldDecl cannot occur because of non-empty newDecls dependencies – if the generation of oldDecl had occurred the method solve would have been called and not our method. Then the KeyInfo waiting for oldDecl now have to wait for the MissingFunctionGeneration::\_waitDeclarations, MissingClassGeneration::\_waitDeclarations that has been visited. These waited declarations are precisely newDecls. The case occurs in the methods VisitTable::setInstanceClassAsComplete, VisitTable::addIncompleteFunction, VisitTable::addIncompleteClass.

* + The methods isMissingDecl, isGenerationMissing and the classes MissingDecl, MissingFunctionGeneration, MissingClassGeneration,
  + the fields MissingFunctionGeneration::\_waitDeclarations, MissingClassGeneration::\_waitDeclarations,
  + the method solve,
  + the methods VisitTable::setInstanceClassAsComplete, VisitTable::addIncompleteFunction, VisitTable::addIncompleteClass.

**virtual** **bool** solve(**const** clang::Decl\* decl, ForwardReferenceList& globals, VisitTable& table);

This method is called on incomplete entries (see the method isGenerationMissing) to notify them that oldDecl has been generated. If it is the last dependency of our entry, then it has to be generated. For this generation we supply the parameter globals. For classes containing subclasses the methods VisitTable::solve and VisitTable::addWaitFor enable to solve the subclass or to generate at least its declaration.

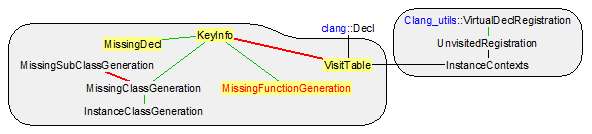
The method is called when oldDecl is visited although it was waited by other (isGenerationMissing()) KeyInfo and when the generation of decl has occurred – if it was not the case, the method replaceWaitingDecl would have been called and not our method. The case occurs in the methods VisitTable::addDeclaration, VisitTable::setInstanceClassAsComplete.

1. The fields MissingFunctionGeneration::\_waitDeclarations and MissingClassGeneration::\_waitDeclarations should contain decl.
   * The method isGenerationMissing and the classes MissingFunctionGeneration, MissingClassGeneration,
   * the fields MissingFunctionGeneration::\_waitDeclarations, MissingClassGeneration::\_waitDeclarations,
   * the method replaceWaitingBy,
   * the methods VisitTable::addDeclaration, VisitTable::setInstanceClassAsComplete.

The class MissingFunctionGeneration

The class MissingFunctionGeneration contains the visit info available for a clang::FunctionDecl that is an instance of template and such that one or many template arguments have not been visited. The translation\_unit\_decl is soon built when the constructor is called. But its Cabs generation in the global ForwardReferenceList is conditioned to the visit (and the generation) of the missing declarations \_waitDeclarations.

The inheritance graph of this class is defined as following.



On the “Figure 3: simple example of template code”, the generation of class X after the visit of Bar creates two MissingFunctionGeneration waiting for Bar2, one for the destructor X::~X() and one for the method X::setPointer. As soon as Bar2 is visited, the MissingFunctionGeneration are translated into pure KeyInfo.

Fields of the class MissingFunctionGeneration

translation\_unit\_decl \_waitingFunDefinition;

Cabs function body. Its generation in the global ForwardReferenceList is conditioned to the visit of the clang declarations present in \_waitDeclarations. This field is not **nullptr** and is defined by the constructor.

std::vector<**const** clang::Decl\*> \_waitDeclarations;

This field defines the clang declarations that are waited for the generation of the function body. This field is not empty and is set up manually by VisitTable each time a MissingFunctionDecl is created, in particular in the methods VisitTable::addIncompleteFunction, VisitTable::addWaitFor.

Declaration of the class MissingFunctionGeneration

**class** MissingFunctionGeneration : **public** KeyInfo {

**private**:

translation\_unit\_decl \_waitingFunDefinition;

std::vector<**const** clang::Decl\*> \_waitDeclarations;

**friend** **class** VisitTable;

**public**:

MissingFunctionGeneration(**const** clang::FunctionDecl\* key, translation\_unit\_decl waitingDefinition)

: KeyInfo(key), \_waitingFunDefinition(waitingDefinition) {}

**virtual** ~MissingFunctionGeneration()

{ **if** (\_waitingFunDefinition) { free\_translation\_unit\_decl(\_waitingFunDefinition); \_waitingFunDefinition = **NULL**; }; }

**virtual** **bool** isComplete() **const** { **return** !\_waitingFunDefinition && \_waitDeclarations.empty(); }

**virtual** **bool** isGenerationMissing() **const** { **return** **true**; }

**virtual** **bool** isFunctionGenerationMissing() **const** { **return** **true**; }

**virtual** **bool** solve(**const** clang::Decl\* decl, ForwardReferenceList& globals, VisitTable& table);

**virtual** **void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls);

};

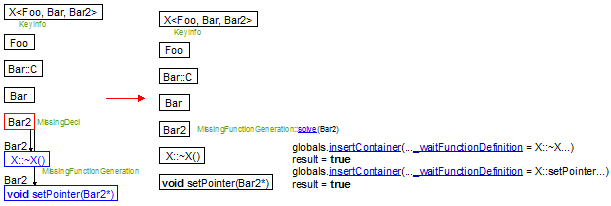
Methods of the class MissingFunctionGeneration

Public methods

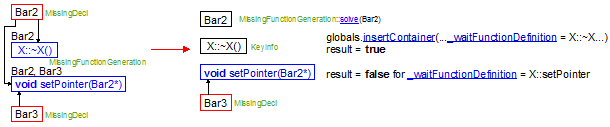
**virtual** **bool** solve(**const** clang::Decl\* decl, ForwardReferenceList& globals, VisitTable& table);

This method is called on our template instance function to notify it that decl has been generated, according to the specification given in KeyInfo::solve.

On the example Figure 3, the method has the following behavior:



The declaration solving may be partial like on the following schema:



1. The field \_waitDeclarations should contain decl.
2. The field \_waitDeclarations should have removed decl.
   * The field \_waitDeclarations,
   * the method replaceWaitingBy,
   * the methods MissingClassGeneration::solve, MissingSubClassGeneration::removeWaiting, VisitTable::solve,
   * the methods VisitTable::addDeclaration, VisitTable::setInstanceClassAsComplete.

**virtual** **void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls);

This method is called on our template instance function to replace the dependence to oldDecl with the new dependence newDecls, according to the specification given in KeyInfo::replaceWaitingBy.

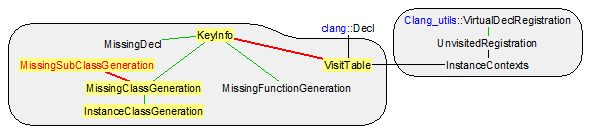
The implementation does nothing but replaces oldDecl by newDecls in \_waitDeclarations viewed as a set of clang::Decl. The method MissingClassGeneration::replaceWaitingBy provides an equivalent schema.

1. The field \_waitDeclarations should contain oldDecl and newDecls should not be empty.
2. The field \_waitDeclarations does not contain oldDecl but all newDecls in one exemplary.
   * The fields \_waitDeclarations,
   * the method solve,
   * the methods MissingClassGeneration::replaceWaitingBy, MissingSubClassGeneration::replaceWaitingBy,
   * the methods VisitTable::setInstanceClassAsComplete, VisitTable::addIncompleteFunction, VisitTable::addIncompleteClass.

The class MissingSubClassGeneration

The class MissingSubClassGeneration contains the visit info available for the content of a clang::RecordDecl that is an instance of template and such that one or many template arguments have not been visited. The class\_decl is soon built when the constructor is called. It is a branch of the translation\_unit\_decl carried by the top MissingClassGeneration and ready to be generated. Two cases are likely to occur. If our MissingSubClassGeneration finally depends on the same last parameter than its top MissingClassGeneration, then it simply forgets the generation of \_waitingSubClassDecl since its top MissingClassGeneration has done the job. In the other cases, \_subWaitDeclarations is not empty when the top MissingClassGeneration generation occurs in the global ForwardReferenceList and our MissingSubClassGeneration is translated into a MissingClassGeneration with its own MissingClassGeneration::\_waitDeclarations – see the method VisitTable::addWaitFor.

The inheritance graph of this class is defined as following.



On the “Figure 3: simple example of template code”, the generation of class X after the visit of Bar creates two MissingFunctionGeneration waiting for Bar2, one for the destructor X::~X() and one for the method X::setPointer. As soon as Bar2 is visited, the MissingFunctionGeneration are translated into pure KeyInfo.

Fields of the class MissingSubClassGeneration

**const** clang::Decl\* \_key;

This field represents the clang declaration we are waiting for its generation. This key is used to find the MissingSubClassGeneration within the fields MissingSubClassGeneration::\_subGenerations and MissingClassGeneration::\_subGenerations. We do not use set but a vector because in a given class there is usually a small number of sub-classes.

This key is not **nullptr**. It should be present in the fields MissingClassGeneration::\_subGenerations, MissingClassGeneration::\_subWaitDeclarations or in the fields MissingSubClassGeneration::\_subGenerations, MissingSubClassGeneration::\_subWaitDeclarations of its parent.

class\_decl \_waitingSubClassDecl;

This field is the Cabs part that waits for the visit of its top MissingClassGeneration and for the visit of the declarations in \_additionalWaitDeclarations to be generated. \_waitingSubClassDecl is a subpart of its top tree MissingClassGeneration::\_waitingClassDeclaration. \_waitingSubClassDecl is **nullptr** if the method removeWait has emptied \_additionalWaitDeclarations.

If \_additionalWaitDeclarations is empty when the generation of MissingClassGeneration::\_waitingClassDeclaration occurs, then we simply forget this field. If it is not the case, we disconnect \_waitingSubClassDecl from the top MissingClassGeneration::\_waitingClassDeclaration and we generate a new MissingClassGeneration with \_waitingSubClassDecl as its waiting field.

std::vector<**const** clang::Decl\*> \_additionalWaitDeclarations;

Sometimes the sub-class is templated or it depends on sub-arguments of the template instance that are not required for the top class generation. In that case \_additionalWaitDeclarations records these additional dependencies.

This field is the Cabs part that waits for the visit of its top MissingClassGeneration and for the visit of the declarations in \_additionalWaitDeclarations should not be empty at the MissingSubClassGeneration construction but it can become empty after many calls to the function removeWait.

std::vector<MissingSubClassGeneration> \_subGenerations;

As nested classes exist, our construction can be one and it can contain sub-elements that are waiting for different declarations that the one required for the generation of our class.

std::set<**const** clang::Decl\*> \_subWaitDeclarations;

This field is a summary of all keys present in \_subGenerations. Hence we quickly know how to look for a particular clang::Decl. If it is not present in our field we just have no need to look into \_subGenerations.

We have some invariants:

* \_subWaitDeclarations is the summary of all keys present in \_subGenerations.
* The fields \_waitingSubClassDecl in \_subGenerations are accessible (sub-trees) from our \_waitingSubClassDecl if it is defined.
* \_waitingSubClassDecl = **nullptr** ⇔ \_additionalWaitDeclarations = ∅.
* The intersection is empty between the declarations present in \_additionalWaitDeclarations and in \_subGenerations.

Declaration of the class MissingSubClassGeneration

**class** MissingSubClassGeneration {

**private**:

**const** clang::Decl\* \_key;

class\_decl \_waitingSubClassDecl;

std::vector<**const** clang::Decl\*> \_additionalWaitDeclarations;

std::vector<MissingSubClassGeneration> \_subGenerations;

std::set<**const** clang::Decl\*> \_subWaitDeclarations;

**friend** **class** VisitTable;

**public**:

MissingSubClassGeneration(**const** clang::RecordDecl\* key, class\_decl waitingSubClassDecl)

: \_key(key), \_waitingSubClassDecl(waitingSubClassDecl) {}

**void** addWaitFor(**const** clang::Decl\* decl) { \_additionalWaitDeclarations.push\_back(decl); }

MissingSubClassGeneration& createSubDeclaration(**const** clang::RecordDecl\* key, class\_decl waitingSubClassDecl)

{ \_subGenerations.push\_back(MissingSubClassGeneration(key, waitingSubClassDecl)); **return** \_subGenerations.back(); }

std::vector<**const** clang::Decl\*>& waitDeclarations() { **return** \_additionalWaitDeclarations; }

**bool** removeWait(**const** clang::Decl\* decl);

**void** setAsComplete() { **assert**(\_waitingSubClassDecl); **if** (\_additionalWaitDeclarations.empty()) \_waitingSubClassDecl = **nullptr**; }

**void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls);

};

Methods of the class MissingSubClassGeneration

Public methods

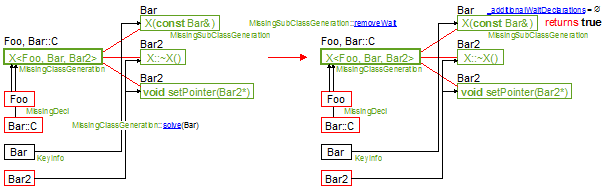
**bool** removeWait(**const** clang::Decl\* decl);

This method notifies that decl has been visited and generated (see the method KeyInfo::solve).

This method suppresses a declaration from \_additionalWaitDeclarations or recursively from one of our \_subGenerations. It returns **true** if and only if \_additionalWaitDeclarations and \_subGenerations are empty after the suppression. In that case the caller can delete our MissingSubClassGeneration since the generation of \_waitingSubClassDecl is now handled by its parent.

This method is called by MissingClassGeneration::solve when decl is a dependency of MissingClassGeneration::\_subWaitDeclarations.

This method performs the following action on this simple case.



1. decl is present in \_additionalWaitDeclarations or \_subGenerations.
2. If this method returns **true**, our MissingSubClassGeneration should be suppressed from the field MissingClassDeclaration::\_subGenerations or MissingSubClassDeclaration::\_subGenerations of its parent.
   * The fields \_additionalWaitDeclarations and \_subGenerations,
   * the method replaceWaitingBy,
   * the method MissingClassGeneration::solve.

**void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls);

This method notifies that oldDecl has been visited but that its generation should wait for the declarations in newDecls.

This method replaces the declaration oldDecl from \_additionalWaitDeclarations by newDecls or recursively from one of our \_subGenerations. Calling this method induces no modification for the caller since the status of its generation has not changed.

This method is called by MissingClassGeneration::replaceWaitingBy when oldDecl is a dependency of MissingClassGeneration::\_subWaitDeclarations.

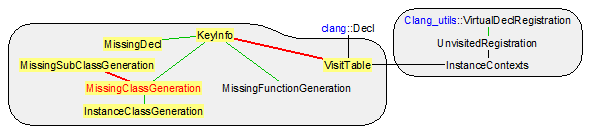
* + The fields \_additionalWaitDeclarations and \_subGenerations,
  + the method removeWait,
  + the method MissingClassGeneration::replaceWaitingBy.

The class MissingClassGeneration

The class MissingClassGeneration contains the visit info available for a clang::RecordDecl that is an instance of template class and such that one or many template arguments have not been visited. The translation\_unit\_decl is soon built when the constructor is called. But its Cabs generation in the global ForwardReferenceList is conditioned to the visit (and the generation) of the missing declarations \_waitDeclarations.

The declarations in this class may have a different status than our MissingClassGeneration since they may depend on different declarations. In that case the field \_subGenerations contains all the declarations MissingSubClassGeneration that have more dependencies than the ones in \_waitDeclarations.

The inheritance graph of this class is defined as following.



On the “Figure 3: simple example of template code”, the class X<Foo, Bar, Bar2> is initially delayed to the visit of the classes Foo and Bar::C. So we create a MissingClassGeneration waiting for Foo and Bar::C. It contains three MissingSubClassGeneration that have additional dependencies to Bar (constructor X::X(**const** Bar& source)) and Bar2 (destructor X::~X() and the method X::setPointer).

The first time we enter in a class instance, we do not know if the generation will be immediate or if it will be delayed. So we create an InstanceClassGeneration and we use InstanceClassGeneration::\_waitingDecls to store the declarations that were in MissingDecl and that are waiting for our class generation. During the visit we collect the dependencies in \_waitDeclarations. At the end of the visit of our class, if some effective dependencies are not solved, we translate our InstanceClassGeneration into a MissingClassGeneration and for each InstanceClassGeneration::\_waitingDecls we replace its dependencies to our class with the dependencies in \_waitDeclarations.

Fields of the class MissingClassDeclaration

translation\_unit\_decl \_waitingClassDeclaration;

Cabs class body. Its generation in the global ForwardReferenceList is conditioned to the visit of the clang declarations present in \_waitDeclarations. This field is not **nullptr** and is defined by the constructor.

std::vector<**const** clang::Decl\*> \_waitDeclarations;

This field defines the clang declarations that are waited for the generation of the class body. This field is not empty and is set up manually by VisitTable each time a MissingClassDecl is created, in particular in the methods VisitTable::addIncompleteClass, VisitTable::addWaitFor. Note that some sub-declarations in the class may depend on additional waited clang declarations. The field \_subGenerations should contain all such sub-declarations.

std::vector<MissingSubClassGeneration> \_subGenerations;

This field contains the sub-declarations of our class that are waiting for different clang declarations that the ones \_waitDeclarations required for the generation of our class.

std::set<**const** clang::Decl\*> \_subWaitDeclarations;

This field is a summary of all keys present in \_subGenerations. Hence we quickly know how to look for a particular clang::Decl. If it is not present in our field we just have no need to look into \_subGenerations.

We have some invariants:

* \_subWaitDeclarations is the summary of all keys present in \_subGenerations.
* The fields MissingSubClassDeclaration::\_waitingSubClassDecl in \_subGenerations are accessible (sub-trees) from our \_waitingClassDeclaration.
* The intersection is empty between the declarations present in \_waitDeclarations and in \_subGenerations.

Declaration of the class MissingClassDeclaration

**class** MissingClassGeneration : **public** KeyInfo {

**private**:

translation\_unit\_decl \_waitingClassDeclaration;

std::vector<**const** clang::Decl\*> \_waitDeclarations;

std::vector<MissingSubClassGeneration> \_subGenerations;

std::set<**const** clang::Decl\*> \_subWaitDeclarations;

**friend** **class** VisitTable;

**public**:

MissingClassGeneration(**const** clang::RecordDecl\* key, translation\_unit\_decl waitingDeclaration)

: KeyInfo(key), \_waitingClassDeclaration(waitingDeclaration) {}

**virtual** ~MissingClassGeneration()

{ **if** (\_waitingClassDeclaration) { free\_translation\_unit\_decl(\_waitingClassDeclaration); \_waitingClassDeclaration = **nullptr**; }; }

MissingSubClassGeneration& createSubDeclaration(**const** clang::RecordDecl\* key, class\_decl waitingSubClassDecl)

{ \_subGenerations.push\_back(MissingSubClassGeneration(key, waitingSubClassDecl)); **return** \_subGenerations.back(); }

std::vector<**const** clang::Decl\*>& waitDeclarations() { **return** \_waitDeclarations; }

**virtual** **bool** isClassGenerationMissing() **const** { **return** **true**; }

**virtual** **bool** isGenerationMissing() **const** { **return** **true**; }

**virtual** **bool** isComplete() **const**

{ **return** !\_waitingClassDeclaration && \_waitDeclarations.empty() && \_subGenerations.empty() && \_subWaitDeclarations.empty(); }

**virtual** **bool** solve(**const** clang::Decl\* decl, ForwardReferenceList& globals, VisitTable& table);

**virtual** **void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls);

};

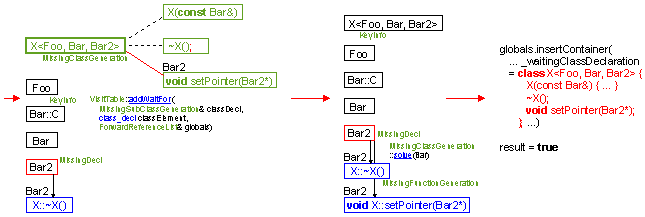
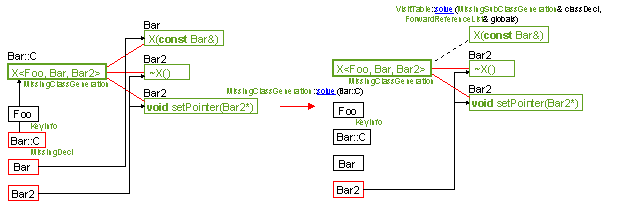
Methods of the class MissingClassDeclaration

Public methods

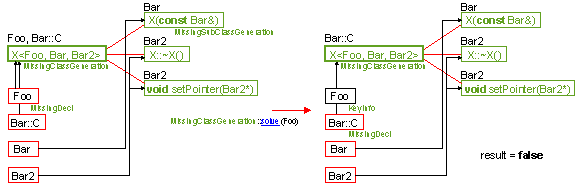
**virtual** **bool** solve(**const** clang::Decl\* decl, ForwardReferenceList& globals, VisitTable& table);

This method is called on our template instance class to notify it that decl has been generated, according to the specification given in KeyInfo::solve.

On the example Figure 3, the method has the following behavior:



The declaration solving may be partial like on the following schema:



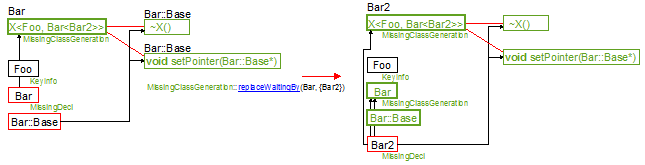
decl may be present only in one or several sub-declarations present in \_subGenerations. We know if we are in such a case if \_subWaitDeclarations contains decl. In that case we call the method MissingSubClassGeneration::removeWait on the sub-declarations that depends on decl to remove this dependency. As specified in MissingSubClassGeneration::removeWait, the MissingSubClassGeneration is suppressed from \_subGenerations if this method returns **true**.

1. Either the field \_waitDeclarations contains decl or \_subWaitDeclarations contains decl.
2. The field \_waitDeclarations should have removed decl or \_subWaitDeclarations should have removed all the dependencies to decl.
   * The field \_waitDeclarations,
   * the methods VisitTable::solve, VisitTable::addWaitFor, MissingSubClassGeneration::removeWait and the constructors of the classes MissingFunctionGeneration, MissingClassGeneration,
   * the method replaceWaitingBy,
   * the methods MissingFunctionGeneration::solve,
   * the methods VisitTable::addDeclaration, VisitTable::setInstanceClassAsComplete.

**virtual** **void** replaceWaitingBy(**const** clang::Decl\* oldDecl, **const** std::vector<**const** clang::Decl\*>& newDecls);

This method is called on our template instance function to replace the dependence to oldDecl with the new dependence newDecls, according to the specification given in KeyInfo::replaceWaitingBy.

The implementation does nothing but replaces oldDecl by newDecls in \_waitDeclarations viewed as a set of clang::Decl. On the example Figure 3, the method has the following behavior:



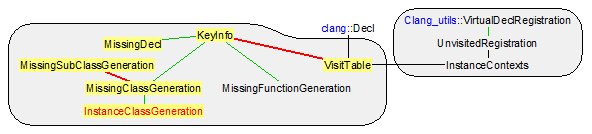
This method may call recursively MissingSubClassGeneration::replaceWaitingBy if oldDecl is not in \_waitDeclarations but in \_subGenerations.

1. Either the field \_waitDeclarations contains oldDecl or \_subGenerations (and so \_subWaitDeclarations) contains oldDecl. newDecls should not be empty.
2. The field \_waitDeclarations does not contain any more reference to oldDecl, nor \_subWaitDeclarations. If \_waitDeclarations contained oldDecl, it now contains all newDecls in one exemplary.
   * The fields \_waitDeclarations, \_subGenerations, \_subWaitDeclarations and the method MissingSubClassGeneration::replaceWaitingBy,
   * the method solve,
   * the methods MissingFunctionGeneration::replaceWaitingBy,
   * the methods VisitTable::setInstanceClassAsComplete, VisitTable::addIncompleteFunction, VisitTable::addIncompleteClass.

The class InstanceClassGeneration

The class InstanceClassGeneration is a MissingClassGeneration whose lifetime is limited to the visit of its corresponding class/record \_key. The first time we enter in a class instance, we do not know if the generation will be immediate or if it will be delayed. So we create an InstanceClassGeneration and we use \_waitingDecls to store the declarations that were in MissingDecl and that are waiting for our class generation. During the visit we collect the dependencies in \_waitDeclarations. At the end of the visit of our class, if some effective dependencies are not solved, the method VisitTable::setInstanceClassAsComplete translate our InstanceClassGeneration into a MissingClassGeneration and for each \_waitingDecls we replace its dependencies to our class with the dependencies in \_waitDeclarations. If all dependencies \_waitDeclarations are solved, the method VisitTable::setInstanceClassAsComplete translates our InstanceClassGeneration into a pure KeyInfo.

The inheritance graph of this class is defined as following:



Fields of the class InstanceClassGeneration

WaitingDecls \_waitingDecls;

This field is used to store the declarations that are waiting for the generation of our class. The storage lifetime is limited to the visit of our corresponding class/record \_key. This field is filled when a MissingDecl is translated into a MissingClassGeneration with a transfer of MissingDecl::\_waitingDecls into our \_waitingDecls. At the end of the visit, for each \_waitingDecls, VisitTable::setInstanceClassAsComplete replaces its dependencies to our class with the dependencies in \_waitDeclarations. Or if \_waitDeclarations is empty, it calls KeyInfo::solve on each waiting declaration of \_waitingDecls.

Declaration of the class InstanceClassGeneration

**class** InstanceClassGeneration : **public** MissingClassGeneration {

**public**:

**typedef** std::vector<KeyInfo\*> WaitingDecls;

**private**:

WaitingDecls \_waitingDecls;

**friend** **class** VisitTable;

**public**:

InstanceClassGeneration(**const** clang::RecordDecl\* key, translation\_unit\_decl waitingDeclaration)

: MissingClassGeneration(key, waitingDeclaration) {}

**virtual** **bool** isInstanceClass() **const** { **return** **true**; }

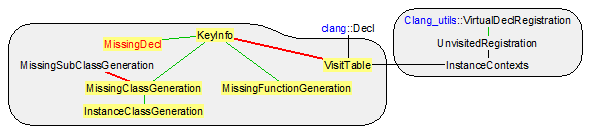
**virtual** **bool** solve(**const** clang::Decl\* decl, ForwardReferenceList& globals, VisitTable& table) { assert(**false**); }

};

The class MissingDecl

The class MissingDecl represents a clang declaration that has not been visited. As it is present in our VisitTable, some visited instances actually need its generation. They are all registered in the field \_waitingDecls. As soon as the visit occurs, our MissingDecl is translated into a pure KeyInfo if its generation is effective. In the other cases (missing declarations for the generation), it is translated into a MissingClassGeneration or a MissingFunctionGeneration, depending on the type of \_key. As the visit is defined by two events: entering in the class and exiting from the class, our MissingDecl is first translated into a InstanceClassGeneration for the enter event. The exit event translates the InstanceClassGeneration into a pure KeyInfo or a MissingClassGeneration, depending whether the generation can occur or not.

The inheritance graph of this class is defined as following:



Fields of the class MissingDecl

WaitingDecls \_waitingDecls;

This field is used to store the declarations that are waiting for the generation of our declaration. Once the declaration is visited the \_waitingDecls are visited. If the visit produces a Cabs generation, all the elements of \_waitingDecls will be KeyInfo::solve. If the visit induces no generation, the elements of \_waitingDecls will be KeyInfo::replaceWaitingBy with the declarations on which our \_key is depending.

Declaration of the class MissingDecl

**class** MissingDecl : **public** KeyInfo {

**public**:

**typedef** std::vector<KeyInfo\*> WaitingDecls;

**private**:

WaitingDecls \_waitingDecls;

**friend** **class** VisitTable;

**public**:

MissingDecl(**const** clang::Decl\* decl) : KeyInfo(decl) {}

**virtual** **bool** isMissingDecl() **const** { **return** **true**; }

**virtual** **bool** isComplete() **const** { **return** **false**; }

WaitingDecls& waitingDecls() { **return** \_waitingDecls; }

};

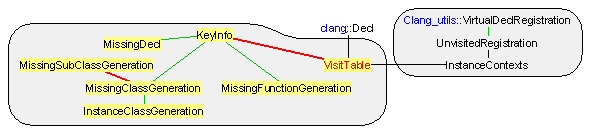
The class VisitTable

The class VisitTable records the information related to the declarations whose visitation has an impact on the template instance generation. Hence all visited declarations (class, function, typedef, constant) should be registered to know if an instance can have access to its definition of if it has to wait for it.

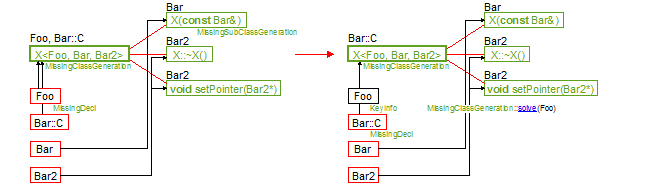
The main field of this class is a map \_content from clang::Decl to visit information on the declaration. 4 types of information KeyInfo are available:

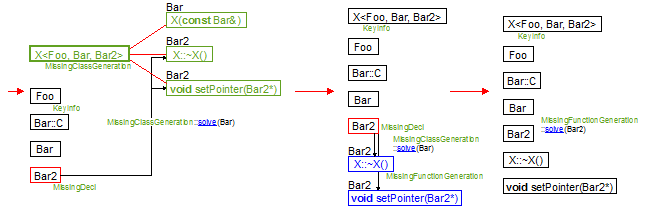
* The name of a declaration has not been encountered. It is represented by the absence of entry in the VisitTable map.
* The name of a declaration has been encountered but not its body. It is represented by a connection clang::Decl → MissingDecl in the map.
* The declaration has been visited but cannot be generated due to missing declarations. It is represented by a connection clang::Decl → MissingFunctionGeneration or clang::Decl → MissingClassGeneration in the map.
* The declaration has been visited and has been generated. It is represented by a connection clang::Decl → KeyInfo in the VisitTable map.

The inheritance graph of this unit is defined on the following schema.

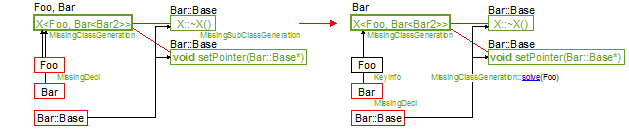


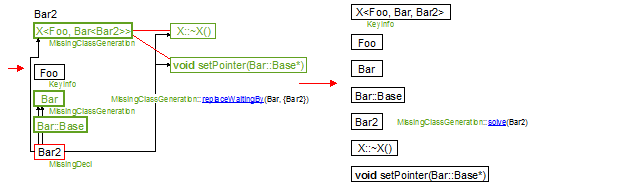
The following sequence of schemas describes the evolution of the VisitTable during the visit of clang declarations in Figure 3: simple example of template code. At the end of the algorithm isComplete returns **true**, which means that all clang declarations have produced their Cabs corresponding.





On the example in Figure 5: variation for the Figure 3 example, the next figure describes the evolution of our VisitTable during the visit of clang declarations. At the end of the algorithm isComplete also returns **true**: all clang declarations have produced their Cabs corresponding.





Fields of the class VisitTable

Clang\_utils\* \_clangUtils;

This field is set up just after the construction of our VisitTable to externalize the declarations intern of a class. On the following example,

|  |  |  |  |
| --- | --- | --- | --- |
| **template** <**class** T1, T2>  **struct** X {  T1\* t1;  T2 t2;  X() : t1(**new** T1) {}  ~X() { **if** (t1) **delete** t1; }  }; | **class** A;  **class** B { … }; | **struct** X<A, B> {  A\* t1;  B t2;  X();  ~X();  }; | **class** A { … };  X<A, B>::X()  : t1(**new** A) {}  X<A, B>::~X()  { **if** (t1) **delete** t1; } |

the generation of the methods of X<A, B> is at the charge of VisitTable and requires to qualify this methods. This is done by calls to Clang\_utils::makeQualifiedName with \_clangUtils.

ContentTable \_content;

Defines the map that associates to each encounter clang::Decl a type of information among the 4 types available:

* The name of a declaration has not been encountered. It is represented by the absence of entry in the VisitTable map.
* The name of a declaration has been encountered but not its body. It is represented by a connection clang::Decl → MissingDecl in the map.
* The declaration has been visited but cannot be generated due to missing declarations. It is represented by a connection clang::Decl → MissingFunctionGeneration or clang::Decl → MissingClassGeneration in the map.
* The declaration has been visited and has been generated. It is represented by a connection clang::Decl → KeyInfo in the VisitTable map.

Declaration of the class VisitTable

**class** VisitTable {

**public**:

**class** KeyInfo;

**class** MissingFunctionGeneration;

**class** MissingSubClassGeneration;

**class** MissingClassGeneration;

**class** InstanceClassGeneration;

**class** MissingDecl;

**private**:

**typedef** std::set<KeyInfo\*, KeyInfo::Less> ContentTable;

Clang\_utils\* \_clangUtils;

ContentTable \_content;

**protected**:

**void** solve(MissingSubClassGeneration& classDecl, ForwardReferenceList& globals);

**void** addWaitFor(MissingSubClassGeneration& classDecl, class\_decl classElement, ForwardReferenceList& globals);

**friend** **class** MissingSubClassGeneration;

**friend** **class** MissingClassGeneration;

**public**:

VisitTable() : \_clangUtils(**nullptr**) {}

~VisitTable() { **for** (KeyInfo\* key : \_content) { **if** (key) **delete** key; }; \_content.clear(); }

**void** setUtils(Clang\_utils\* clangUtils) { \_clangUtils = clangUtils; }

**bool** isComplete() **const** { **for** (KeyInfo\* key : \_content) { **if** (!key->isComplete()) **return** **false**; }; **return** **true**; }

**bool** hasVisited(**const** clang::Decl\* decl) **const**

{ **auto** found = \_content.find(&KeyInfo(decl)); **return** (found != \_content.end()) && !(\*found)->isMissingDecl(); }

**void** addDeclaration(**const** clang::Decl\* decl, ForwardReferenceList& globals);

MissingClassGeneration& addInstanceClass(**const** clang::RecordDecl\* decl, translation\_unit\_decl classDecl);

MissingSubClassGeneration& addSubClass(MissingClassGeneration& firstInstance, MissingSubClassGeneration\* lastClass, **const** clang::RecordDecl\* decl, class\_decl classDecl)

{ **return** (!lastClass) ? firstInstance.createSubDeclaration(decl, classDecl) : lastClass->createSubDeclaration(decl, classDecl); }

**void** setInstanceClassAsComplete(InstanceClassGeneration\* instance, ForwardReferenceList& globals);

MissingClassGeneration& addIncompleteClass(**const** clang::RecordDecl\* decl, std::vector<**const** clang::Decl\*>& waitDeclarations, translation\_unit\_decl classDecl);

MissingFunctionGeneration& addIncompleteFunction(**const** clang::FunctionDecl\* decl, std::vector<**const** clang::Decl\*>& waitDeclarations, translation\_unit\_decl functionDecl);

};

Methods of the class VisitTable

Protected Methods

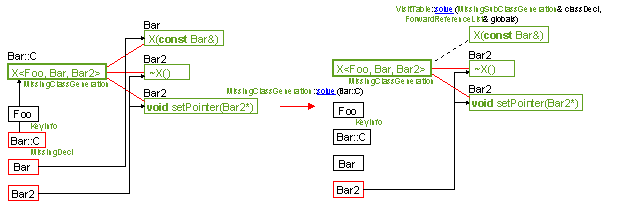
**void** solve(MissingSubClassGeneration& classDecl, ForwardReferenceList& globals);

This method is called on the declaration classDecl.\_key in a class template to notify that this declaration is solved at the same time than its ancestor MissingClassGeneration.

The implementation mainly propagates on classDecl.\_subGenerations – by default, the generation of the outer class generates the inner classes. Each element of classDecl.\_subGenerations that has a MissingSubClassGeneration::\_waitingSubClassDecl should be externalized. On such sub-declaration the algorithm calls MissingSubClassGeneration::addWaitFor. On the other sub-declarations, it recursively calls MissingSubClassGeneration::solve.

The declarations in instances of classes do not appear in \_content except to be associated with a MissingDecl. In this case, we wake up the MissingFunctionGeneration and the MissingClassGeneration depending on classDecl.\_key.

On the example Figure 3, the method has the following behavior:



This method is called by MissingClassGeneration::solve to propagate the outer class generation to the inner classes.

* + The method classDecl.removeWait should have returned **true**,
  + classDecl.\_waitingSubClassDecl == **nullptr**,
  + classDecl.\_additionalWaitDeclarations.empty().
  + The method addWaitFor and the method KeyInfo::solve, MissingClassGeneration::solve, MissingFunctionGeneration::solve,
  + the methods MissingClassGeneration::solve, MissingSubClassGeneration::removeWait,
  + the methods VisitTable::addDeclaration, VisitTable::setInstanceClassAsComplete.

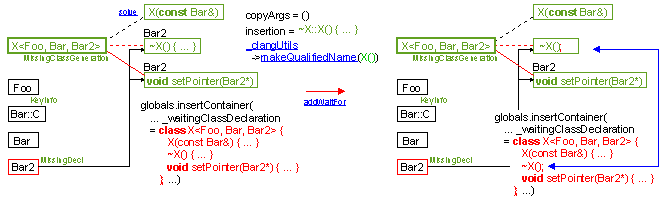
**void** addWaitFor(MissingSubClassGeneration& classDecl, class\_decl classElement, ForwardReferenceList& globals);

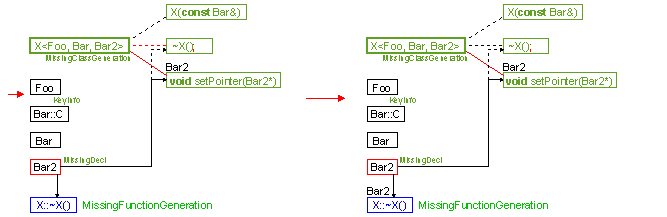
This method transforms the Cabs definition classElement into a declaration. The original Cabs definition is duplicated at the beginning of the call of our method and the copy is externalized and classDecl is translated into a MissingClassDeclaration, waiting for new clang visit to be generated in globals.

As the declaration containing classElement has soon been generated in globals, this generation can wake up new generations depending on classDecl.\_key: the declarations in instances of classes do not appear in \_content except to be associated with a MissingDecl (see the method solve). In this case, we wake up the MissingFunctionGeneration and the MissingClassGeneration depending on classDecl.\_key.

A last point consists in the registration of the newly created MissingClassDeclaration for it to be waked up when the MissingDecl associated to the elements of the old classDecl.\_additionalWaitDeclarations will be visited.

On the example Figure 3, the method has the following behavior:





The method is called by the method solve for the sub-declarations of a MissingSubClassGeneration (which should be a class) that have additional dependencies – MissingSubClassGeneration::\_waitingSubClassDecl ≠ **nullptr** and !MissingSubClassGeneration::\_additionalWaitDeclarations.empty(). If it concerns the sub-declaration classDecl of a MissingClassGeneration, then the method setInstanceClassAsComplete directly calls our method on the sub-declaration that has additional dependencies.

* + classDecl.\_waitingSubClassDecl ≠ **nullptr**,
  + !classDecl.\_additionalWaitDeclarations.empty().
  + The constructors MissingFunctionGeneration::MissingFunctionGeneration, MissingClassGeneration::MissingClassGeneration and the methods MissingClassGeneration::solve, MissingFunctionGeneration::solve,
  + the method solve,
  + the methods VisitTable::setInstanceClassAsComplete, MissingClassGeneration::solve.

Public Methods

**void** addDeclaration(**const** clang::Decl\* decl, ForwardReferenceList& globals);

The method notifies that the non-template decl has been visited and generated. The possible side-effect is the notification to all KeyInfo in \_content that decl is solved. The notification calls the method KeyInfo::solve on each element of MissingDecl::waitingDecls() and the generation occurs if and only if decl was the last dependency of this KeyInfo.

This method is called after the visit of each non-template clang::Decl: the concerned methods are Visitor::postVisitRecordDecl, Visitor::VisitEnumDecl, Visitor::VisitTypedefNameDecl, Visitor::VisitFunctionDecl, Visitor::VisitVarDecl, Visitor::VisitFieldDecl.

1. If decl is referenced in \_content, it should be associated to a MissingDecl.
   * The class MissingDecl and the methods KeyInfo::solve, MissingClassGeneration::solve, MissingFunctionGeneration::solve,
   * the methods setInstanceClassAsComplete, addInstanceClass, addIncompleteFunction,
   * the methods Visitor::postVisitRecordDecl, Visitor::VisitEnumDecl, Visitor::VisitTypedefNameDecl, Visitor::VisitFunctionDecl, Visitor::VisitVarDecl, Visitor::VisitFieldDecl.

MissingClassGeneration& addInstanceClass(**const** clang::RecordDecl\* decl, translation\_unit\_decl classDecl);

The method notifies that the visit enters into a class instance decl. As the visit does not know what the dependent declarations are, it does not know if the generation will be immediate or if it will be delayed. By default our method creates an InstanceClassGeneration and the visit of decl will collect the dependencies in MissingClassGeneration::\_waitDeclarations.

Once the dependencies will be known and solved, the visit should trigger the solving on the MissingFunctionGeneration and on the MissingClassGeneration that depend on decl. That is why our method transfers in InstanceClassGeneration::\_waitingDecls the field MissingDecl::\_waitingDecls that has recorded the dependent KeyInfo of decl before the call to our method.

At the end of the visit of our class, if some effective dependencies are not solved, the method setInstanceClassAsComplete will translate the InstanceClassGeneration result into a MissingClassGeneration and for each InstanceClassGeneration::\_waitingDecls it will replace its dependencies to our class with the dependencies in MissingClassGeneration::\_waitDeclarations. If all dependencies MissingClassGeneration::\_waitDeclarations are solved, the method setInstanceClassAsComplete will translate the InstanceClassGeneration result into a pure KeyInfo.

This method is called by Visitor::VisitRecordDecl on a class instance.

1. If decl is referenced in \_content, it should be associated to a MissingDecl.
   * InstanceContexts::pushInstanceContext has to be called on the result of our method. The reason is that the visit has to fill the dependencies MissingClassGeneration::\_waitDeclarations.
   * The method setInstanceClassAsComplete has to be called at the end of the visit of decl.
   * The classes MissingDecl, InstanceClassGeneration and the fields MissingClassGeneration::\_waitDeclarations, MissingDecl::\_waitingDecls, InstanceClassGeneration::\_waitingDecls,
   * the methods setInstanceClassAsComplete, InstanceContexts::pushInstanceContext, addDeclaration, addIncompleteFunction,
   * the methods Visitor::VisitRecordDecl.

**void** setInstanceClassAsComplete(InstanceClassGeneration\* instance, ForwardReferenceList& globals);

This method notifies that the visit exits from a class instance instance->\_key. It receives as instance the result of the method addInstanceClass. Two cases occur depending on the dependent declarations the visitor has found or not dependencies on unvisited declarations (see UnvisitedDeclarations::registerDecl).

The first case concerns the absence of dependent declarations instance->\_waitDeclarations.empty(). If no unvisited dependent declarations have been found, we generate the class and its content. If the content depends on additional declarations (!MissingSubClassGeneration::\_additionalWaitDeclarations.empty() and MissingSubClassGeneration::\_waitingSubClassDecl ≠ **nullptr**), we call MissingSubClassGeneration::addWaitFor on it. If the content is independent of any declaration, we call MissingSubClassGeneration::solve on it. If there are instances instance‑>\_waitingDecls that are waiting for our instance, we call KeyInfo::solve on them (in fact MissingClassGeneration::solve and MissingFunctionGeneration::solve). At the end we replace instance by a pure KeyInfo to indicate the clang declaration instance->\_key has been visited and generated.

The second case concerns the presence of dependent declarations !instance->\_waitDeclarations.empty(). Then for each clang declaration instance->\_waitDeclarations we are waiting for, we make our instance depend from them and we also make all the instance‑>\_waitingDecls also depend from them.

At the end we call the method KeyInfo::replaceWaitingBy to replace the dependency of instance->\_key by dependencies of instance->\_waitDeclarations on each waiting declaration (MissingClassGeneration or MissingFunctionGeneration) of instance‑>\_waitingDecls. Last but not least, we replace instance by a MissingClassGeneration, to remove the field InstanceClassGeneration::\_waitingDecls which is no more useful.

Our method is called by Visitor::postVisitRecordDecl when the visit exits from a class instance instance->\_key.

* + The method addInstanceClass should have been called when the visit has entered the class instance instance‑>\_key,
  + the method UnvisitedDeclarations::registerDecl may have been called several times during the visit of the declarations in the clang class instance‑>\_key to record the dependencies of our instance in instance‑>\_waitDeclarations.

1. The method InstanceContexts::popInstanceContext should be called after our method.
   * The classes MissingDecl, InstanceClassGeneration, MissingClassGeneration and the fields MissingClassGeneration::\_waitDeclarations, MissingDecl::\_waitingDecls, InstanceClassGeneration::\_waitingDecls, MissingSubClassGeneration::\_additionalWaitDeclarations, MissingSubClassGeneration::\_waitingSubClassDecl,
   * the methods MissingSubClassGeneration::addWaitFor, MissingSubClassGeneration::solve, KeyInfo::solve, MissingClassGeneration::solve, MissingFunctionGeneration::solve, KeyInfo::replaceWaitingBy,
   * the methods addInstanceClass, UnvisitedRegistration::registerDecl, InstanceContexts::popInstanceContext, addDeclaration, addIncompleteFunction,
   * the methods Visitor::postVisitRecordDecl.

MissingClassGeneration& addIncompleteClass(**const** clang::RecordDecl\* decl, std::vector<**const** clang::Decl\*>& waitDeclarations, translation\_unit\_decl classDecl);

This method corresponds to the addIncompleteFunction for class, but it is not used any more due to the particularity of the visitor: it processes with two events: entering and exiting a class instead of one. That is why this method is replaced by the methods addInstanceClass / setInstanceClassAsComplete.

MissingFunctionGeneration& addIncompleteFunction(**const** clang::FunctionDecl\* decl, std::vector<**const** clang::Decl\*>& waitDeclarations, translation\_unit\_decl functionDecl);

The method notifies that the visit has encountered an instance of a template function/method such that one or many arguments are not completely visited at that time. This means that some required declarations will be visited in the future and that this visit will made the generation of functionDecl effective.

This method creates a MissingFunctionGeneration, associates it to decl in \_content and returns it. The result is not really used except in the internal of our class.

Then for each clang declaration waitDeclarations we are waiting for, we make our instance depend from it. If there were instances that were waiting for decl (a MissingDecl was associated to decl in \_content), we also make all the MissingDecl::\_waitingDecls also depend from waitDeclarations. As there is a double linkage between MissingDecl::\_waitingDecls and MissingClassGeneration::\_waitDeclarations or MissingFunctionGeneration::\_waitDeclarations, we call KeyInfo::replaceWaitingBy to replace the dependency from decl by a dependency from waitDeclarations.

This method is called by Visitor::VisitFunctionDecl on a function instance.

* + waitDeclarations should not be empty,
  + the method InstanceContexts::popInstanceFunction should have been called to fill waitDeclarations.
  + The classes MissingDecl, MissingFunctionGeneration and the fields MissingFunctionGeneration::\_waitDeclarations, MissingDecl::\_waitingDecls,
  + the method KeyInfo::replaceWaitingBy,
  + the methods addInstanceClass, setInstanceClassAsComplete, UnvisitedRegistration::registerDecl, InstanceContexts::popInstanceFunction, addDeclaration,
  + the methods Visitor::VisitFunctionDecl.

The ClangVisitor Unit

The class LocalDeclContext

The class LexicalLocalDeclContext

The class SemanticLocalDeclContext

The class DeclContext

The classes UnvisitedRegistration and UnvisitedNameRegistration

This class inherits from Clang\_utils::VirtualDeclRegistration to implement the virtual method registerDecl. When \_visitor visits a clang declaration, the method registerDecl is automatically called and our class delivers the status of this declaration – has been visited or not. It records then the unvisited declarations in the field \_visitor.unvisitedDecls() for them to be available to the methods VisitTable::setInstanceClassAsComplete, VisitTable::addIncompleteFunction.

In this contexts, the role of the class InstanceContexts is to retrieve the unvisited declarations – \_visitor.unvisitedDecls() is InstanceContext::\_currentContext.back().first and to organize the calls to the right methods VisitTable::setInstanceClassAsComplete, VisitTable::addIncompleteFunction at the right level.

The unvisited declarations are separated into two sorts. The first sort represents the declarations that should be “complete” for the generation. The second sort of this first field represents the declarations that have only to be named. In the following code,

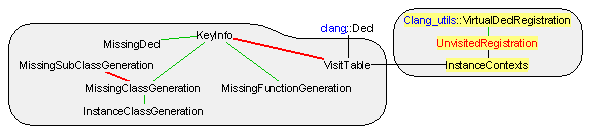
**template** <**class** T, **class** U> **class** A { T\* t; U u; };

A<X, Y> a;

the visit of the instance A<X, Y> requires X to be named and Y to be complete. Such named declarations as X are not stored in a MissingClassGeneration or in a MissingFunctionGeneration, but are to be immediately treated at the end of the visit by the method Visitor::insertNamedDeclaration generating “**class** X;” , called by Visitor::postVisitRecordDecl and Visitor::VisitFunctionDecl.

That is why the method getNameRegistration returns its own field \_unvisitedName that stores unvisited declarations in \_visitor.unvisitedNameDecls() instead of \_visitor.unvisitedDecls().

The inheritance graph of our class is the following:



Fields of the class UnvisitedRegistration

Visitor& \_visitor;

Reference to the current visitor to implement the virtual method registerDecl. This field is set up at the construction of our class.

Declaration of the class UnvisitedRegistration

**class** UnvisitedNameRegistration : **public** Clang\_utils::VirtualDeclRegistration {

**private**:

**typedef** Clang\_utils::VirtualDeclRegistration inherited;

Visitor& \_visitor;

public:

UnvisitedNameRegistration(Visitor& visitor) : \_visitor(visitor) { setRegisterDecl(); }

UnvisitedNameRegistration(**const** UnvisitedNameRegistration& source) : inherited(source), \_visitor(source.\_visitor) {}

**virtual** **void** registerDecl(**const** clang::Decl\* decl)

{ **auto**& unvisited = \_visitor.unvisitedNameDecls();

**if** (!\_visitor.\_tableForWaitingDeclarations.hasVisited(decl))

**if** (std::find\_if(unvisited.begin(),unvisited.end(), (**auto** unvisitedDecl)[decl]{ **return** decl == unvisitedDecl; }) != unvisited.end())

unvisited.push\_back(decl);

};

}

Visitor& getVisitor() **const** { **return** \_visitor; }

};

**class** UnvisitedRegistration : **public** Clang\_utils::VirtualDeclRegistration {

**private**:

**typedef** Clang\_utils::VirtualDeclRegistration inherited;

UnvisitedNameRegistration \_unvisitedName;

**public**:

UnvisitedRegistration(Visitor& visitor) : \_unvisitedName(visitor) { setRegisterDecl(); }

UnvisitedRegistration(**const** UnvisitedRegistration& source) : inherited(source), \_unvisitedName(source.\_unvisitedName) {}

**virtual** **void** registerDecl(**const** clang::Decl\* decl)

{ **if** (!\_unvisitedName.getVisitor().\_tableForWaitingDeclarations.hasVisited(decl))

\_unvisitedName.getVisitor().unvisitedDecls().push\_back(decl);

}

**virtual** VirtualDeclRegistration\* getNameRegistration() { **return** &\_unvisitedName; }

};

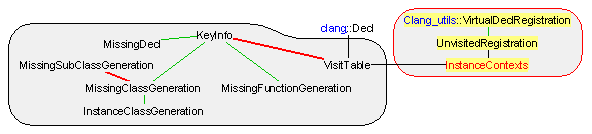
The class InstanceContexts

This unit controls the way the class VisitTable is managed. This unit reacts to many events in particular during the visit of the instance of a class. An object of type InstanceContexts is available in the field Visitor::\_instanceContexts. It manages the other Visitor’s field Visitor::\_tableForWaitingDeclarations.

The main class of this unit is InstanceContexts. It acts as a state machine whose states are:

1. out of any instance and any template – InstanceContexts::\_currentContext.empty() and InstanceContexts::\_waitDeclarationsFunctions.get() = **nullptr**.
2. instance of a template function or a template method – InstanceContexts::\_currentContext.size() = 1 and InstanceContexts::\_waitDeclarationsFunctions.get() ≠ **nullptr**.
3. content of the first instance of a class – InstanceContexts::\_currentContext.size() = 1 and InstanceContexts::\_waitDeclarationsFunctions.get() = **nullptr**.
4. method in an instance of a template class – InstanceContexts::\_currentContext.size() ≥ 2 and InstanceContexts::\_waitDeclarationsFunctions.get() ≠ **nullptr**.
5. class in an instance of a template class – InstanceContexts::\_currentContext.size() ≥ 2 and InstanceContexts::\_waitDeclarationsFunctions.get() = **nullptr**.

The following inheritance graph is used for this unit:

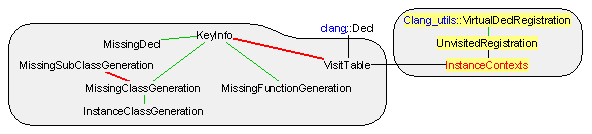


This class controls the way the class VisitTable is managed via the field Visitor::\_tableForWaitingDeclarations. It reacts to many events in particular during the visit of the instance of a class or during the visit of the instance of the body of a function. An object of type InstanceContexts is available in the field Visitor::\_instanceContexts.

The class InstanceContexts acts as a state machine whose states are:

1. out of any instance and any template – \_currentContext.empty() and \_waitDeclarationsFunctions.get() = **nullptr**. Entering a class instance goes to state 3 (see the method push(VisitTable::MissingClassGeneration&)). Entering a function instance goes to state 2 (see the method pushFunction).
2. instance of a template function or a template method – \_currentContext.size() = 1 and \_waitDeclarationsFunctions.get() ≠ **nullptr**. Exiting a function instance goes to state 1 (see the method popFunction).
3. content of the first instance of a class – \_currentContext.size() = 1 and \_waitDeclarationsFunctions.get() = **nullptr**. Entering a class instance goes to state 5 (see the method push(VisitTable::MissingSubClassGeneration&)). Entering a function instance goes to state 4 (see the method pushFunction). Exiting the class instance goes to state 1 (see the method pop).
4. method in an instance of a template class – \_currentContext.size() ≥ 2 and \_waitDeclarationsFunctions.get() ≠ **nullptr**. Exiting the method goes to state 3 or to state 5 (see the method popFunction).
5. class in an instance of a template class – \_currentContext.size() ≥ 2 and \_waitDeclarationsFunctions.get() = **nullptr**. Entering a class instance goes to state 5 (see the method push(VisitTable::MissingSubClassGeneration&)). Entering a function instance goes to state 4 (see the method pushFunction). Exiting the class instance goes to state 3 or to state 5 (see the method pop).

The inheritance graph of our class is the following:



Fields of the class InstanceContexts

std::vector<std::pair<UnvisitedBodyName, LocalContext> > \_currentContext;

Stack of the instances. The stack is required because a class instance can have subclasses that depend on different declarations. The first field corresponds to the clang::Decl that are unknown during the visit of the class. This first field is separated into two sorts. The first sort of this first field represents the declarations that should be “complete” for the generation. The second sort of this first field represents the declarations that have only to be named. The second field depends on the type of the declaration we are visiting: if it is a function, this second field is a LocalContext(); if it is a class instance out of any other class instance, this second field is a LocalContext(VisitTable::MissingClassGeneration\*); if it is a class instance in another class instance, this second field is a LocalContext(VisitTable::MissingSubClassGeneration\*);

Just a note concerning the second sort of the first field, that are the declarations that have only to be named. Such declarations are not stored in a MissingClassGeneration or in a MissingFunctionGeneration. So we do not reference this field but we own it. The declarations that have to be named are immediately treated at the end of the visit by the method Visitor::insertNamedDeclaration, called by Visitor::postVisitRecordDecl and Visitor::VisitFunctionDecl.

std::auto\_ptr<std::vector<**const** clang::Decl\*> > \_waitDeclarationsFunctions;

This field is the owner of the UnvisitedDecls that is at the top of \_currentContext when the last encountered declaration is a function or a method instance. This owner is necessary for functions/methods since VisitTable::addIncompleteFunction works in one step, while VisitTable::addInstanceClass/VisitTable::setInstanceClassAsComplete have two steps, needing to store their own UnvisitedDecls in InstanceClassGeneration::\_waitDeclarations.

The main invariant of the class is the fact that \_currentContext and \_waitDeclarationsFunctions are in state 1, …, state 5. This invariant could be defined only on \_currentContext since \_waitDeclarationsFunctions is valid if and only if \_currentContext.back().second = LocalContext().

Declaration of the class InstanceContexts

**class** InstanceContexts {

**public**:

**typedef** std::vector<**const** clang::Decl\*> UnvisitedDecls;

**private**:

**union** LocalContext {

VisitTable::MissingClassGeneration\* classContent;

VisitTable::MissingSubClassGeneration\* subclassContent;

LocalContext() { classContent = **nullptr**; }

LocalContext(VisitTable::MissingClassGeneration\* content) { classContent = content; }

LocalContext(VisitTable::MissingSubClassGeneration\* content) { subclassContent = content; }

LocalContext(**const** LocalContext& source) { memcpy(**this**, &source, **sizeof**(LocalContext)); }

LocalContext& **operator**=(**const** LocalContext& source) { memcpy(**this**, &source, **sizeof**(LocalContext)); **return** \***this**; }

};

**typedef** std::pair<UnvisitedDecls\*, UnvisitedDecls> UnvisitedBodyName;

std::vector<std::pair<UnvisitedBodyName, LocalContext> > \_currentContext;

std::auto\_ptr<std::vector<**const** clang::Decl\*> > \_waitDeclarationsFunctions;

**public**:

InstanceContexts() {}

**void** push(VisitTable::MissingClassGeneration& context)

{ assert(\_currentContext.empty());

\_currentContext.push\_back(std::make\_pair(std::make\_pair(&context.waitDeclarations(), UnvisitedDecls()), LocalContext(&context)));

}

**void** push(VisitTable::MissingSubClassGeneration& context)

{ assert(!\_currentContext.empty());

\_currentContext.push\_back(std::make\_pair(std::make\_pair(&context.waitDeclarations(), UnvisitedDecls()), LocalContext(&context)));

}

**void** pop() { \_currentContext.pop\_back(); }

**void** pop(std::vector<**const** clang::Decl\*>& namedDeclarations)

{ \_currentContext.back().first.second.swap(namedDeclarations); \_currentContext.pop\_back(); }

**void** pushFunction()

{ assert(!\_waitDeclarationsFunctions.get());

\_waitDeclarationsFunctions.reset(**new** std::vector<**const** clang::Decl\*>());

\_currentContext.push\_back(std::make\_pair(std::make\_pair(&\*\_waitDeclarationsFunctions, UnvisitedDecls()), LocalContext()));

}

**void** popFunction(std::vector<**const** clang::Decl\*>& waitDeclarations, std::vector<**const** clang::Decl\*>& namedDeclarations)

{ assert(\_waitDeclarationsFunctions.get() && waitDeclarations.empty());

\_currentContext.back().first.second.swap(namedDeclarations);

\_waitDeclarationsFunctions->swap(waitDeclarations);

\_waitDeclarationsFunctions.reset();

\_currentContext.pop\_back();

}

**int** size() **const** { **return** \_currentContext.size(); }

**bool** isClassContext() **const** { **return** \_currentContext.size() == 1 && !\_waitDeclarationsFunctions.get(); }

**bool** isSubClassContext() **const** { **return** \_currentContext.size() > 1 && !\_waitDeclarationsFunctions.get(); }

**bool** isEmpty() **const** { **return** \_currentContext.empty() && !\_waitDeclarationsFunctions.get(); }

UnvisitedDecls& unvisitedDecls() { assert(\_currentContext.size() >= 1); **return** \*\_currentContext.back().first.first; }

UnvisitedDecls& unvisitedNameDecls() { assert(\_currentContext.size() >= 1); **return** \_currentContext.back().first.second; }

VisitTable::MissingClassGeneration& lastClassContext() { assert(\_currentContext.size() == 1); **return** \*\_currentContext.back().second.classContent; }

VisitTable::MissingSubClassGeneration\* lastSubClassContext()

{ assert(\_currentContext.size() >= 1); **return** \_currentContext.size() == 1 ? **nullptr** : \_currentContext.back().second.subclassContent; }

VisitTable::MissingClassGeneration& firstClassContext() { assert(\_currentContext.size() >= 1); **return** \*\_currentContext.front().second.classContent; }

};

The class AnnotationCommentList

The class AnnotationCommentHandler

The class LoopAnnotationOption

The class Visitor

The Parser Unit

The class ACSLComment

The class Error

The class AbstractToken

The class CommentToken

The class IdentifierToken

The class KeywordToken

The class LiteralToken

The class IntegerLiteralToken

The class CharacterLiteralToken

The class FloatingLiteralToken

The class StringLiteralToken

The class OperatorPunctuatorToken

The class Token

The class TTextBuffer

The class Lexer

The class RuleResult

The template class TParseState

The template class TStateStack

The class ErrorMessage

The class Arguments

The class Parser

The ACSL++ Unit

The class TermOrPredicate::Operator

The class TermOrPredicate::SubstitutionLevel

The class TermOrPredicate::Substitution

The class TermOrPredicateMemoryExtension

The class TermOrPredicateList

The class Binders

The class WithConstruct

The class Range

The class SetComprehension

The class TermOrPredicate

The class LogicType

The class CodeAnnotation

The class StatementAnnotation

The class LoopAnnotation

The class AssignsClause

The class FunctionContract

The class GlobalAnnotation

The class GlobalAnnotation::Parameters

1. clang: a C language family frontend for LLVM. http://clang.llvm.org. [↑](#footnote-ref-1)
2. The LLVM Compiler Infrastructure. http://www.llvm.org. [↑](#footnote-ref-2)
3. see <http://llvm.org/docs/DeveloperPolicy.html#license> [↑](#footnote-ref-3)
4. see <http://clang.llvm.org/features.html#license> [↑](#footnote-ref-4)