The InstanceTemplate Generation Library

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 1: 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;
};
```

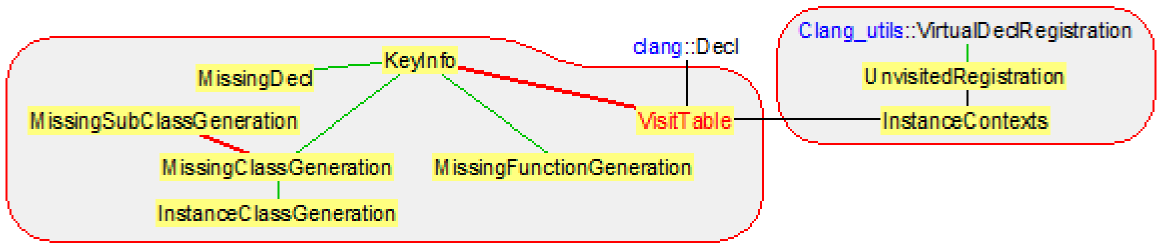
```
class X<Foo, Bar, Bar2>
  : public Foo {
private:
  Bar::C _field;
  Bar2* _pointer;

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 2: generated Cabs for template code instances

The following inheritance graph is used for the implementation:

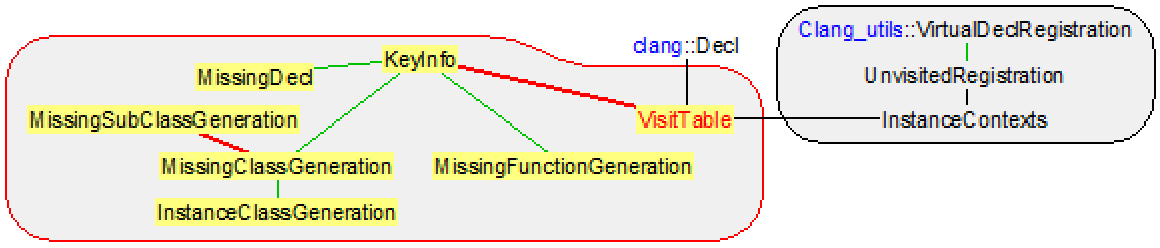


The InfoInstanceTable Unit

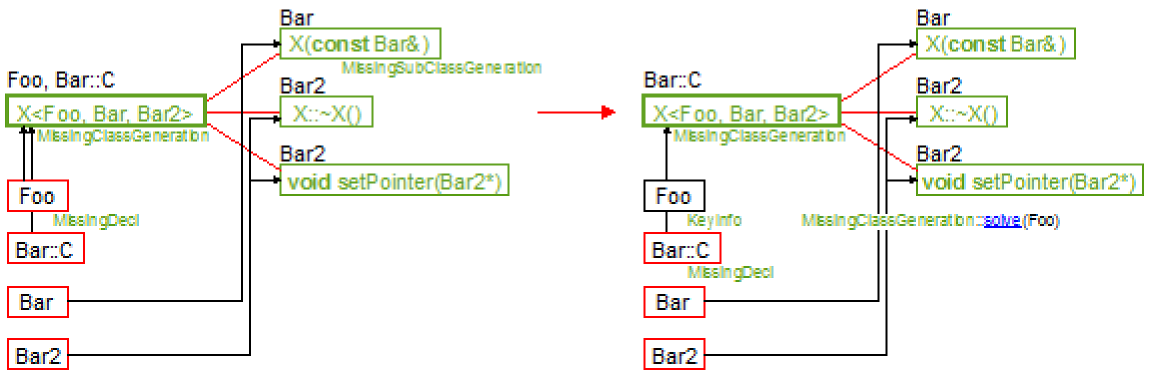
This unit contains the information related to the declarations whose visitation has an impact on the template instance generation. Hence all visited declarations (class, function, typedef, constant) should be registered to know if an instance can have access to its definition of if it has to wait for 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:

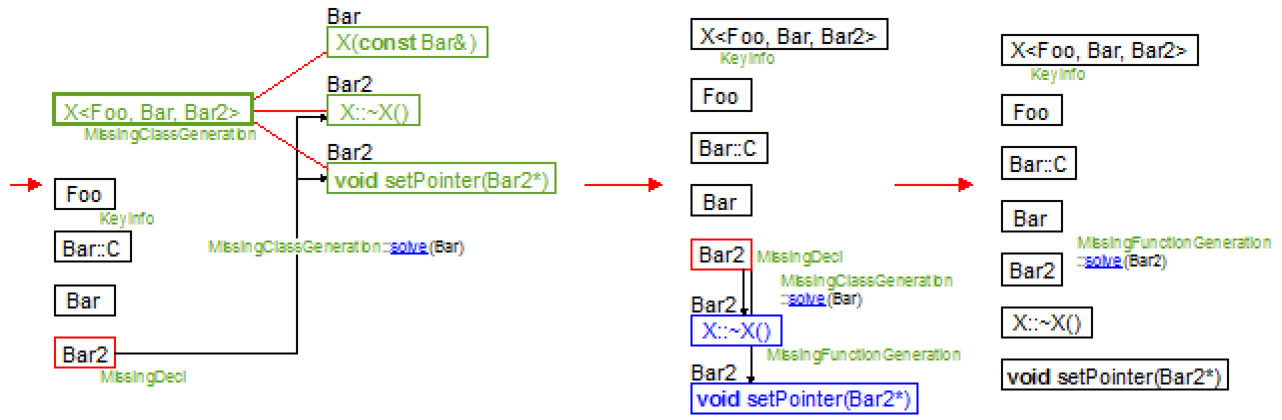
- The name of a declaration has not been encountered. It is represented by the absence of entry in the VisitTable map.
- The name of a declaration has been encountered but not its body. It is represented by a connection clang::Decl → MissingDecl in the VisitTable map.
- The declaration has been visited 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 1: 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.



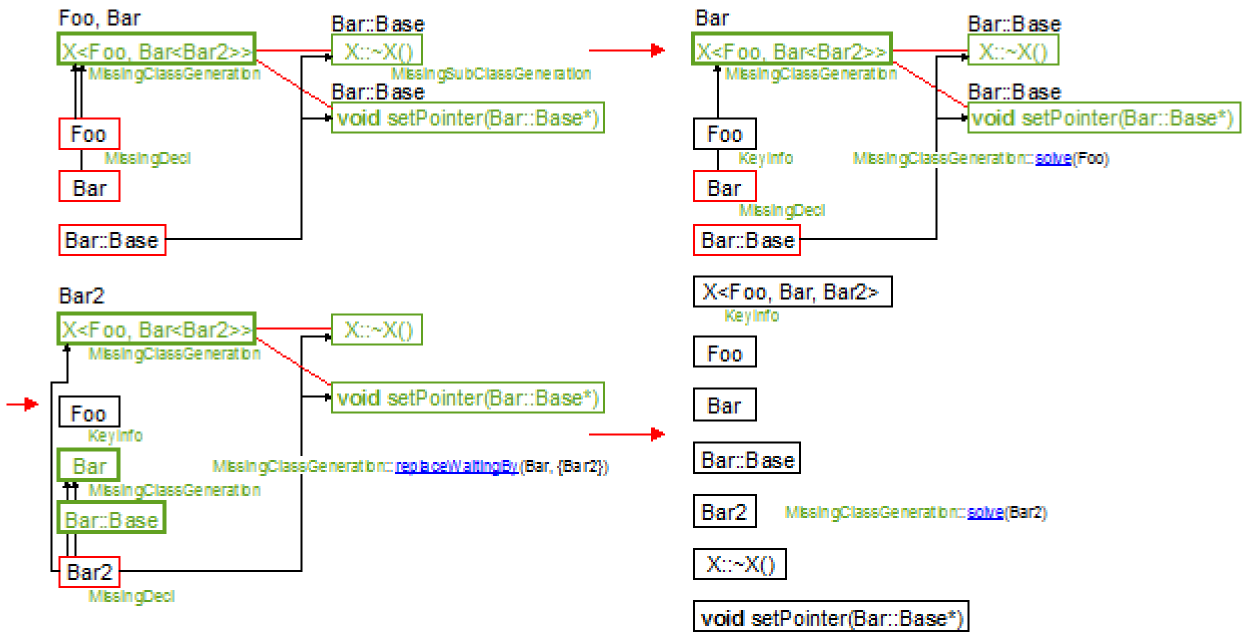


To illustrate another point of the generation algorithm, let us introduce the following example that causes partial instance and the call to KeyInfo::replaceWaitingBy.

```
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 3: variation for the Figure 1 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.

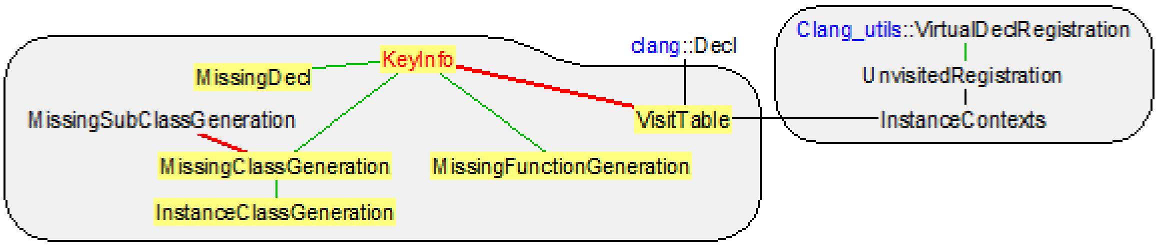


The class KeyInfo

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 1: 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).

Post-conditions: If the method returns **true**, our KeyInfo supports the type MissingDecl.

See also:

- 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.

Post-conditions: If the method returns **true**, you should call isFunctionGenerationMissing or isClassGenerationMissing to know if our KeyInfo supports the type MissingFunctionGeneration or a MissingClassGeneration.

See also:
- 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.

Post-conditions: If the method returns **true**, our KeyInfo supports the type MissingClassGeneration.

See also:
- 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.

Post-conditions: If the method returns **true**, our KeyInfo supports the type InstanceClassGeneration.

Post-conditions:
- 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.

Post-conditions: If the method returns **true**, our KeyInfo supports the type MissingFunctionGeneration.

See also:
- 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.

See also:
- 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.

See also:
- 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.

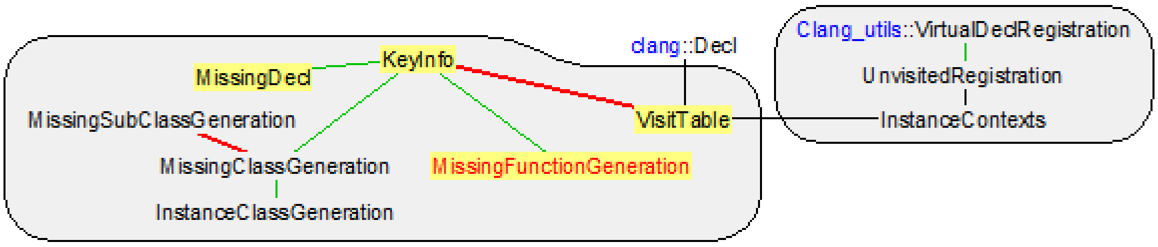
Pre-conditions: The fields MissingFunctionGeneration::_waitDeclarations and MissingClassGeneration::_waitDeclarations should contain decl.

See also:
- 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 1: 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);
};
```

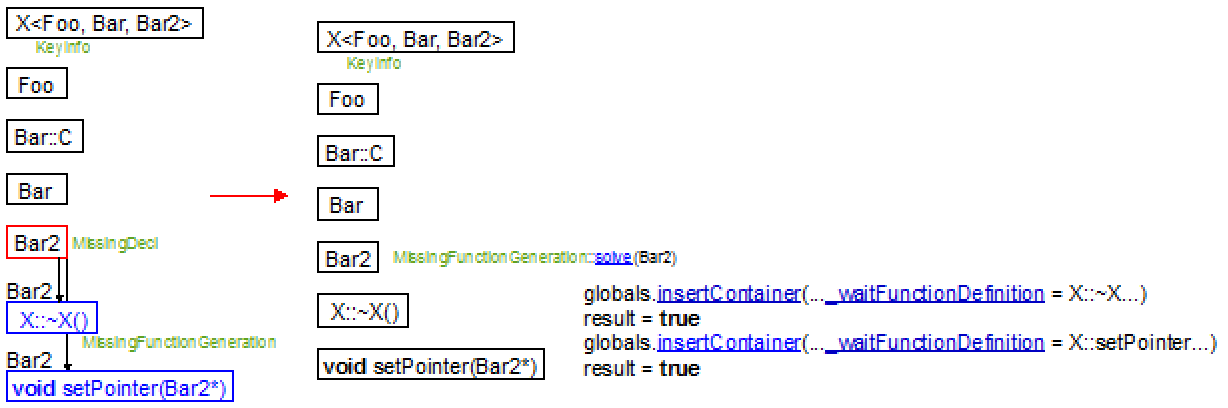
Methods of the class MissingFunctionGeneration

Public methods

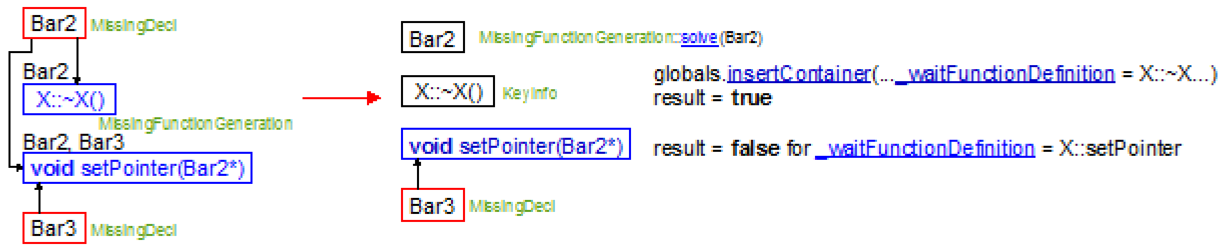
virtual bool solve(const clang::Decl* decl, ForwardReferenceList& globals, VisitTable& table);

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 1, the method has the following behavior:


X<Foo, Bar, Bar2>
KeyInfo
Foo
Bar::C
Bar
Bar2
MissingDecl
Bar2
X::~X()
MissingFunctionGeneration
Bar2
void setPointer(Bar2*)
X<Foo, Bar, Bar2>
KeyInfo
Foo
Bar::C
Bar
Bar2
MissingFunctionGeneration::solve(Bar2)
X::~X()
void setPointer(Bar2*)
globals.insertContainer(..._waitFunctionDefinition = X::~X...)
result = true
globals.insertContainer(..._waitFunctionDefinition = X::setPointer...)
result = true

The declaration solving may be partial like on the following schema:


Bar2
MissingDecl
Bar2
X::~X()
MissingFunctionGeneration
Bar2, Bar3
void setPointer(Bar2*)
Bar3
MissingDecl
Bar2
MissingFunctionGeneration::solve(Bar2)
X::~X()
KeyInfo
globals.insertContainer(..._waitFunctionDefinition = X::~X...)
result = true
void setPointer(Bar2*)
result = false for _waitFunctionDefinition = X::setPointer
Bar3
MissingDecl

Pre-conditions: The field _waitDeclarations should contain decl.

Post-conditions: The field _waitDeclarations should have removed decl.

See also:

- 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.

Pre-conditions: The field _waitDeclarations should contain oldDecl and newDecls should not be empty.

Post-conditions: The field _waitDeclarations does not contain oldDecl but all newDecls in one exemplary.

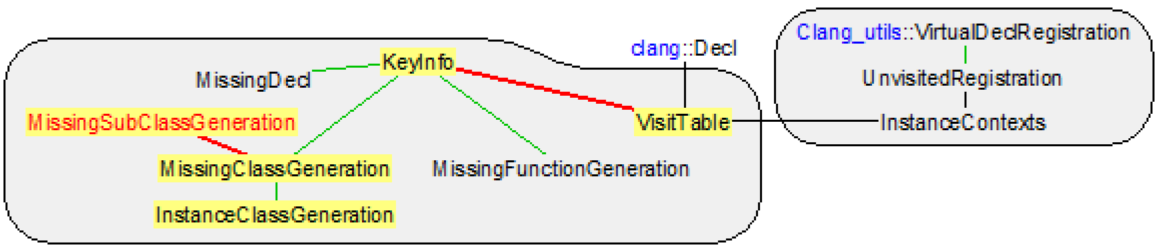
See also:

- 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 1: 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 replaceWaitingBy(const clang::Decl* oldDecl, const std::vector<const clang::Decl*>& newDecls);
};
```

Methods of the class MissingSubClassGeneration

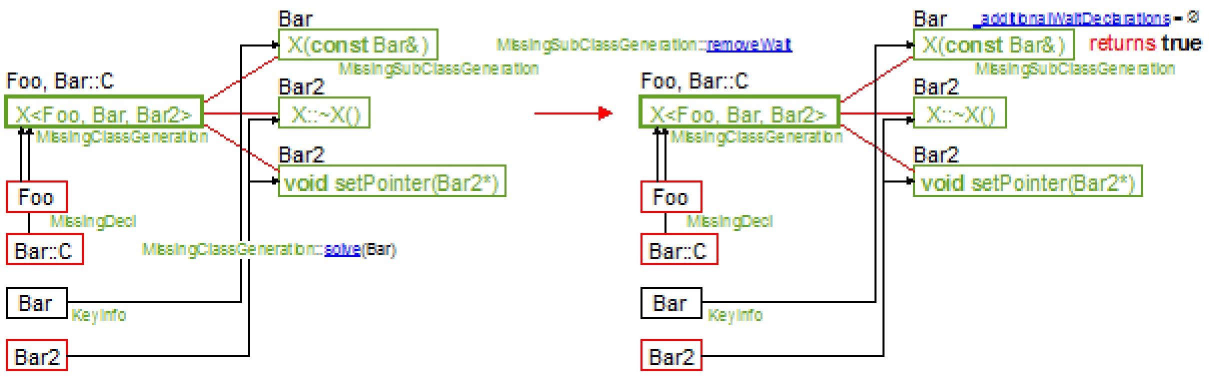
Public methods

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.



Pre-conditions: decl is present in _additionalWaitDeclarations or _subGenerations.

Post-conditions: If this method returns **true**, our MissingSubClassGeneration should be suppressed from the field MissingClassDeclaration::_subGenerations or MissingSubClassDeclaration::_subGenerations of its parent.

See also:

- 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.

See also:

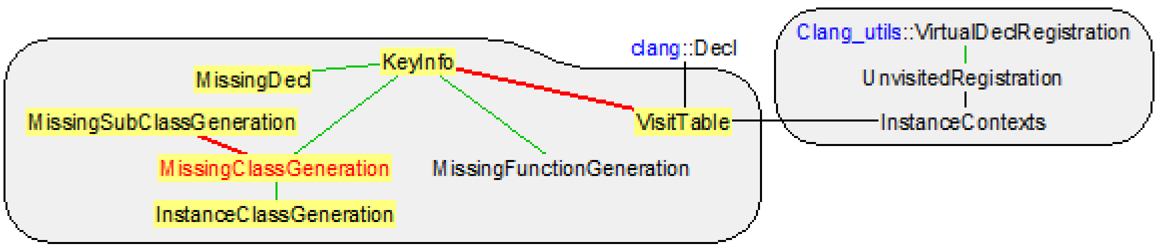
- The fields _additionalWaitDeclarations and _subGenerations,
- the method removeWait,
- the method MissingClassGeneration::replaceWaitingBy.

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 1: 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);
};
```

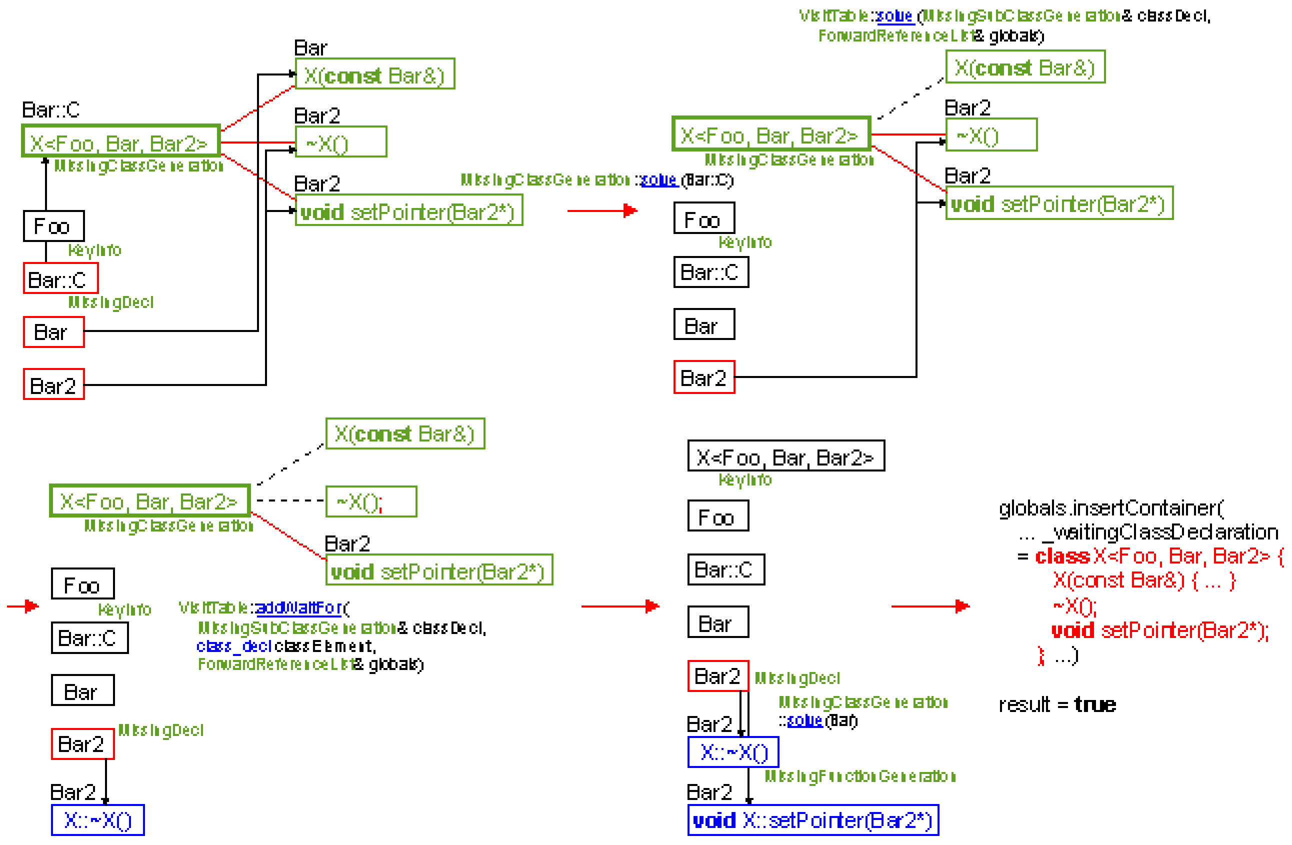
Methods of the class MissingClassDeclaration

Public methods

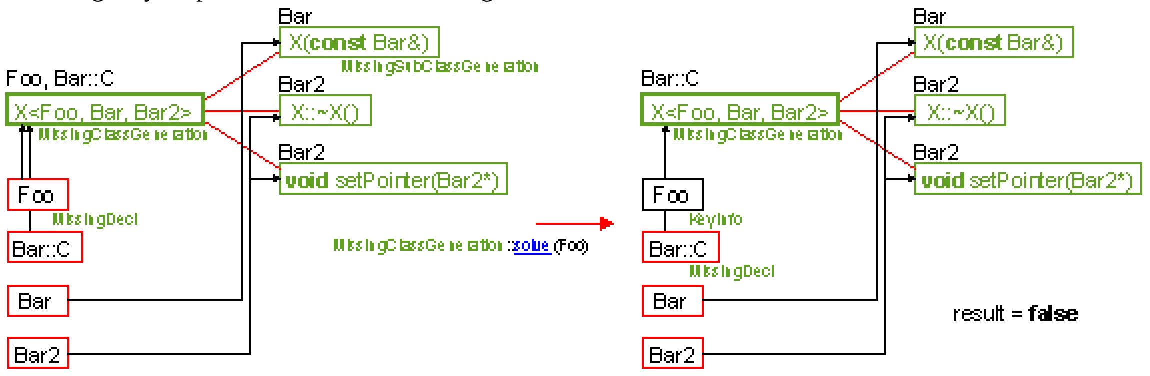
virtual bool solve(**const** clang::Decl* decl, ForwardReferenceList& globals, VisitTable& table);

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 1, 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**.

Pre-conditions: Either the field _waitDeclarations contains decl or _subWaitDeclarations contains decl.

Post-conditions: The field _waitDeclarations should have removed decl or _subWaitDeclarations should have removed all the dependencies to decl.

See also:

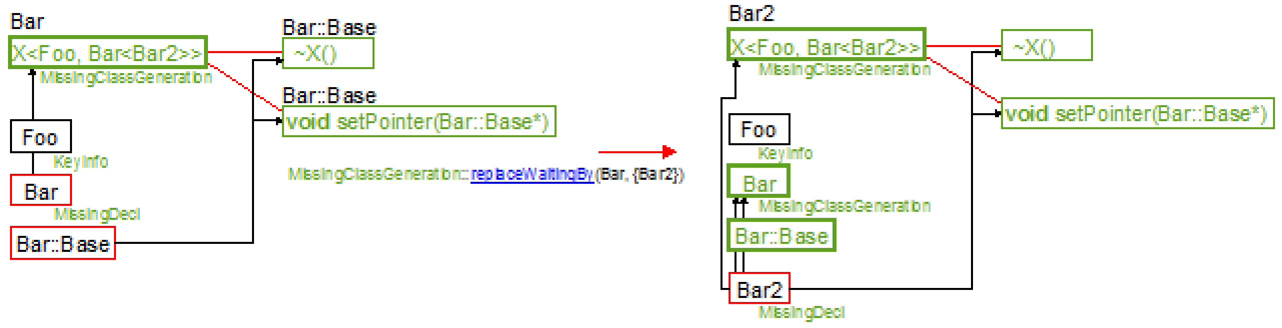
- The field _waitDeclarations,
- the methods VisitTable::solve, VisitTable::addWaitFor, MissingSubClassGeneration::removeWait and the constructors of the classes MissingFunctionGeneration, MissingClassGeneration,
- the method replaceWaitingBy,
- the methods MissingFunctionGeneration::solve,
- the methods 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 1, the method has the following behavior:



This method may call recursively MissingSubClassGeneration::replaceWaitingBy if oldDecl is not in _waitDeclarations but in _subGenerations.

Pre-conditions: Either the field _waitDeclarations contains oldDecl or _subGenerations (and so _subWaitDeclarations) contains oldDecl. newDecls should not be empty.

Post-conditions: 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.

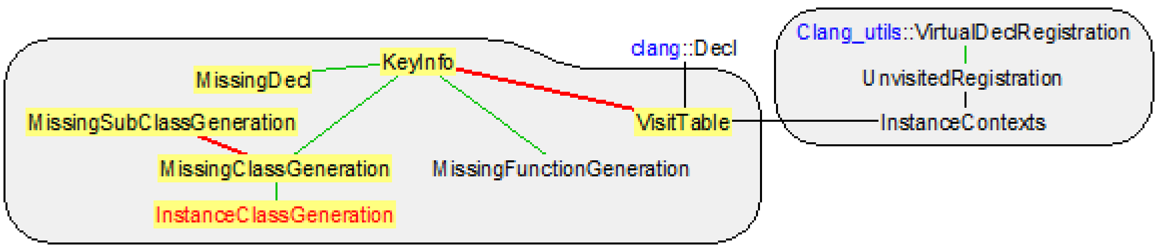
See also:

- 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.

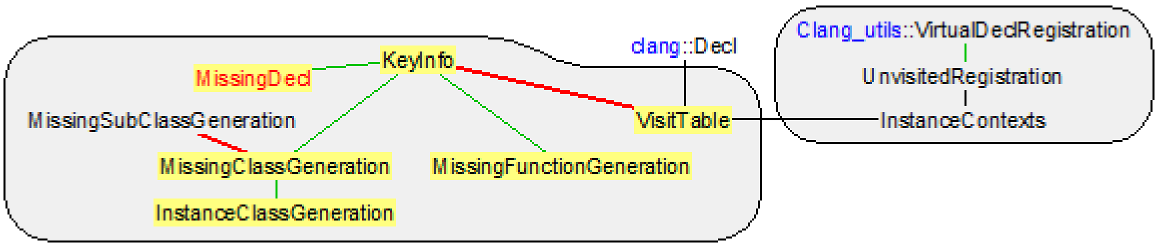
Declaration of the class InstanceClassGeneration

```
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; }
};
```

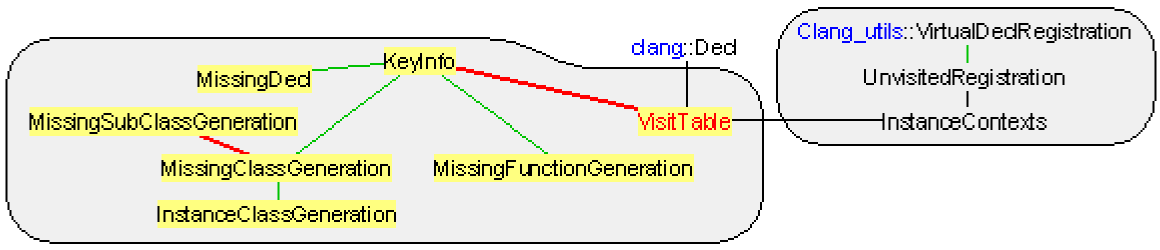
The class VisitTable

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.

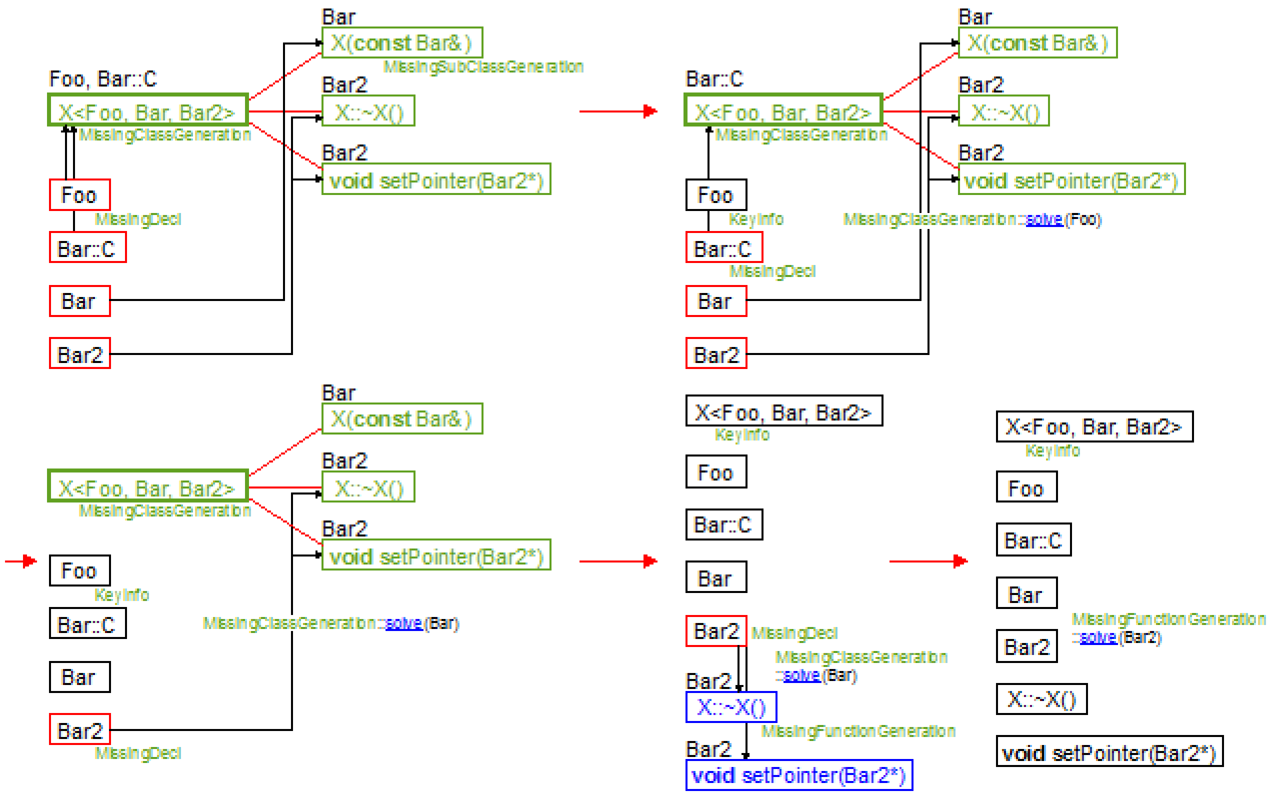
The main field of this class is a map _content from clang::Decl to visit information on the declaration. 4 types of information KeyInfo are available:

- 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.

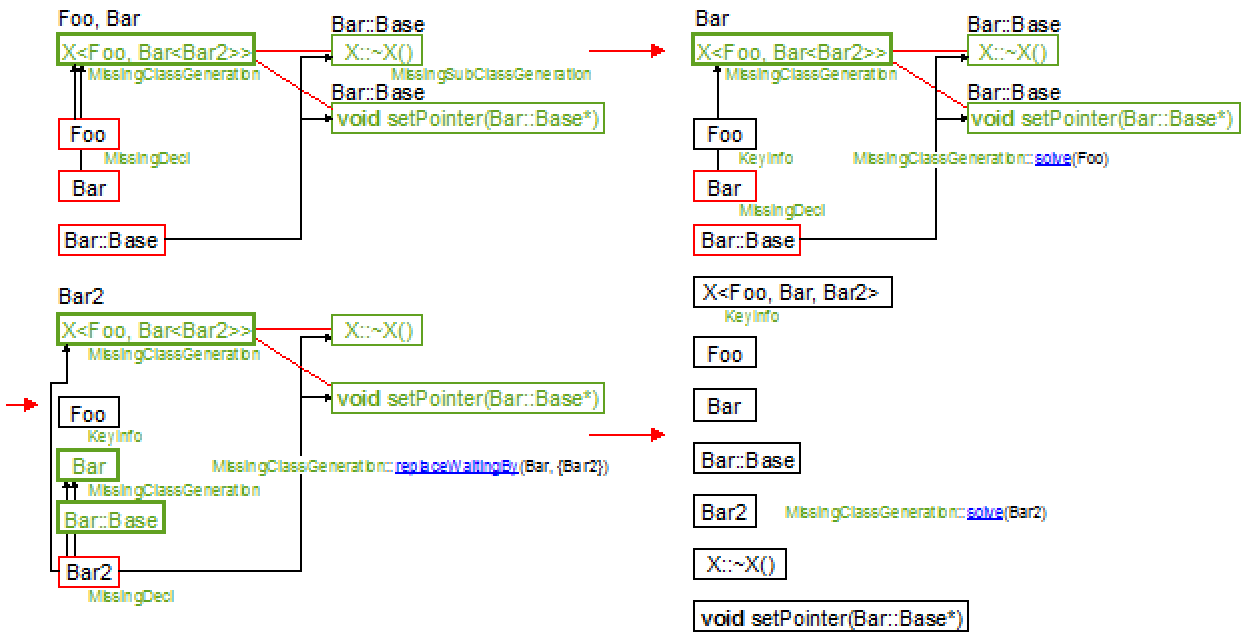
The inheritance graph of this unit is defined on the following schema.



The following sequence of schemas describes the evolution of the VisitTable during the visit of clang declarations in Figure 1: 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.


Foo, Bar::C
X<Foo, Bar, Bar2>
MissingClassGeneration
Bar
X(const Bar&)
MissingSubClassGeneration
Bar2
X::~X()
Bar2
void setPointer(Bar2*)
Foo
MissingDecl
Bar::C
Bar
Bar2
Bar::C
KeyInfo
MissingClassGeneration::solve(Foo)
MissingClassGeneration::solve(Bar)
MissingDecl
MissingClassGeneration::solve(Bar)
MissingFunctionGeneration
MissingFunctionGeneration::solve(Bar2)

On the example in Figure 3: variation for the Figure 1 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

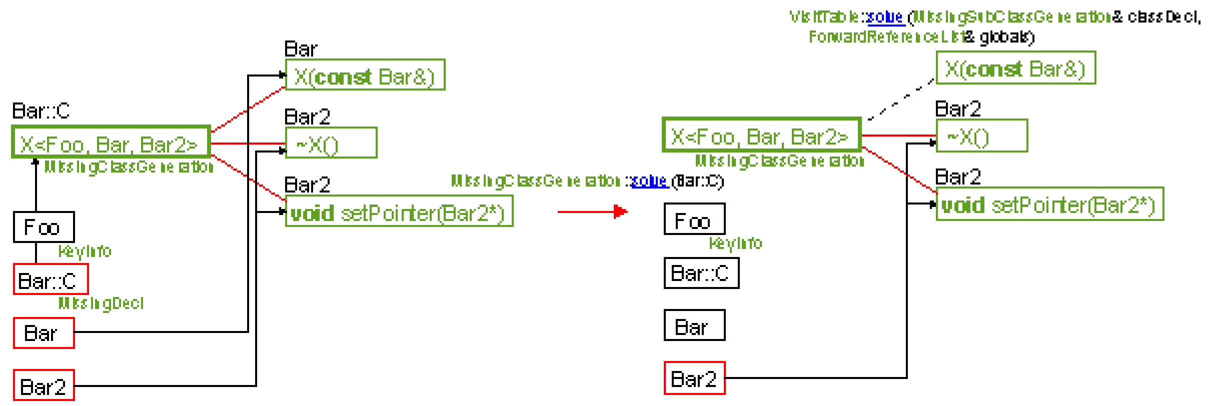
Protected Methods

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 1, the method has the following behavior:



This method is called by MissingClassGeneration::solve to propagate the outer class generation to the inner classes.

Pre-conditions:

- The method classDecl.removeWait should have returned **true**,
- classDecl._waitingSubClassDecl == **nullptr**,
- classDecl._additionalWaitDeclarations.empty().

See also:

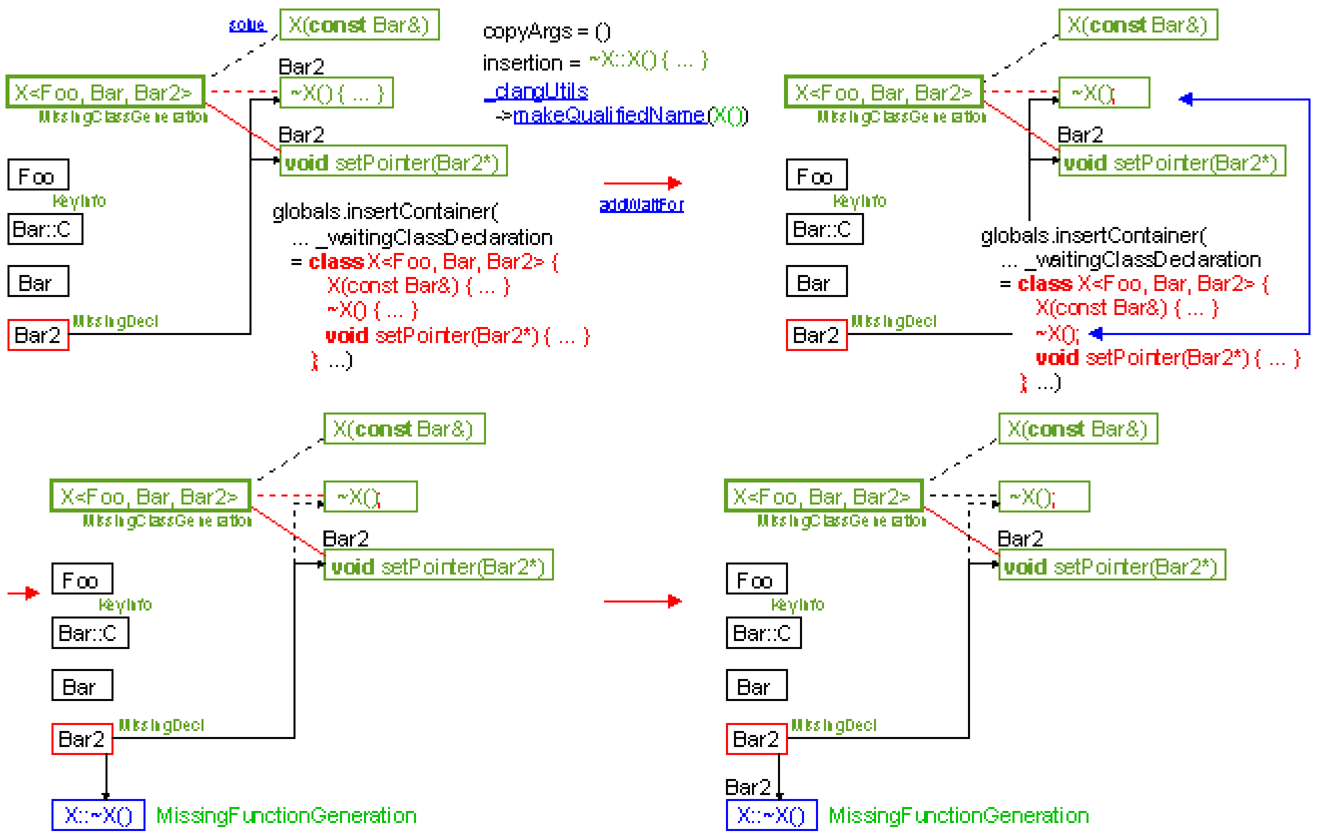
- The method addWaitFor and the method KeyInfo::solve, MissingClassGeneration::solve, MissingFunctionGeneration::solve,
- the methods MissingClassGeneration::solve, MissingSubClassGeneration::removeWait,
- the methods VisitTable::addDeclaration, VisitTable::setInstanceClassAsComplete.

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 1, 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.

Pre-conditions:

- classDecl._waitingSubClassDecl ≠ **nullptr**,
- !classDecl._additionalWaitDeclarations.empty().

See also:

- 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.

Pre-conditions: If decl is referenced in _content, it should be associated to a MissingDecl.

See also:

- 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.

Pre-conditions: If decl is referenced in _content, it should be associated to a MissingDecl.

Post-conditions:
- 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.

See also:
- 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.

Pre-conditions:
- 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.

Post-conditions: The method InstanceContexts::popInstanceContext should be called after our method.

See also:
- 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.

Pre-conditions:
- waitDeclarations should not be empty,
- the method InstanceContexts::popInstanceFunction should have been called to fill waitDeclarations.

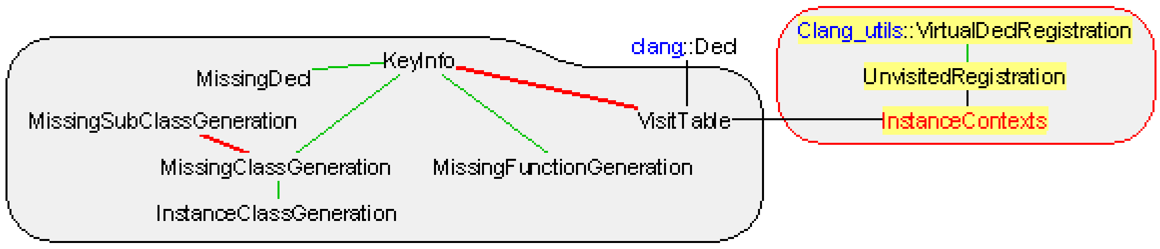
See also:
- The classes MissingDecl, MissingFunctionGeneration and the fields MissingFunctionGeneration::_waitDeclarations, MissingDecl::_waitingDecls,
- the method KeyInfo::replaceWaitingBy,
- the methods addInstanceClass, setInstanceClassAsComplete, UnvisitedRegistration::registerDecl, InstanceContexts::popInstanceFunction, addDeclaration,
- the methods Visitor::VisitFunctionDecl.

The InstanceContexts Unit

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:



The class UnvisitedRegistration

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.

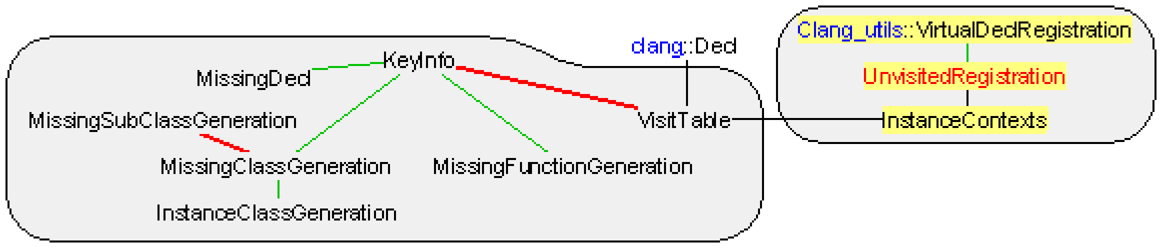
In this contexts, the role of the class InstanceContexts is to retrieve the unvisited declarations – _visitor.unvisitedDecls() is InstanceContext::_currentContext.back().first and to organize the calls to the right methods VisitTable::setInstanceClassAsComplete, VisitTable::addIncompleteFunction at the 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; }
};
```

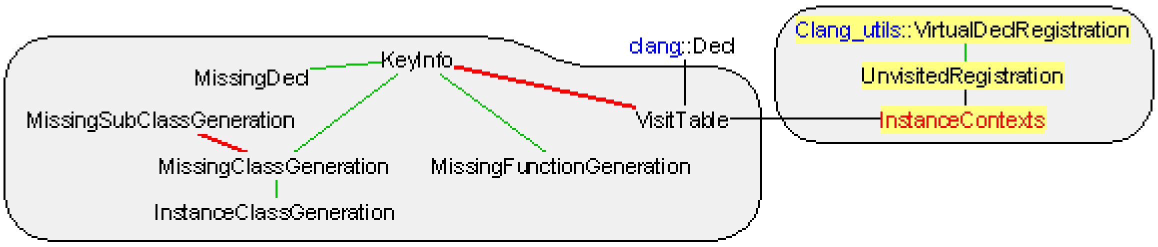
The class InstanceContexts

This 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; }
};
```