Feature Design Specification

CDT 2.0 Parser Symbol Table

Table of Contents

History

Version	*Author*	*Date*	*Description*
0.1	Andrew Niefer	01/23/04	Initial Draft

Introduction

Language Support

Types

(ANSI C 6.2.5) _Complex, _Imaginary, and _Bool types need to be handled by the symbol table. Conversion of these types to other floating and integer types must be considered during function resolution.

GNU extensions make the _Complex and _Imaginary types available in C++.

ANSI C and GNU extensions to C++ allow for a 64 bit long long int which also needs to be supported by the symbol table.

The following flags will be added to the `TypeInfo` class: `isComplex`, `isImaginary`, and `isLongLong`. The "`t__Bool`" type will also be added.

Anonymous Structures & Unions

(ANSI C++ 9.5) A union of the form

```
union { member-specification } ;
```

is called an anonymous union; it defines an unnamed object of unnamed type. For the purposes of name lookup, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared.

Because a union for which objects or pointers are declared is not an anonymous union, the parser won't know until it reaches the end of the union whether or not it is anonymous. The symbol table will provide a function to convert a normal union into an anonymous union once this information is known. The following will be added to IContainerSymbol:

```
void convertToAnonymous();
void isAnonymous();
```

GNU extends this functionality to unnamed struct fields within structs/unions.

C Scoping Rules

(bug46246) In C, there is one namespace for the tags of structures, unions and enumerations. This means that they need to be handled differently from how they are handled in C++ when they are nested.

Eg:

```
struct A {
   struct B { int ab; }; b;
   int a;
};
struct A a1;
struct B b1;
```

Address of Overloaded Functions

(ANSI C++ 13.4) A use of an overloaded function name without arguments is resolved in certain contexts to a function, a pointer to function, or a pointer to member function for a specific function from the overload set. The function selected is the one whose type matches the target type required in the context. The target can be:

- an object or reference being initialized
- the left side of an assignment
- a parameter of a function
- a parameter of a user-defined operator
- the return value of a function, operator function or conversion
- an explicit type conversion.

The following function will be added to `IContainerSymbol` to support this:

```
IParameterizedSymbol targetedFunctionLookup(TypeInfo targetType,
                                            String name );
```

Symbol Iterator

The symbol table must provide an iterator over its contents so that the symbols can be retrieved in the order in which they were added. There are 3 different kinds of symbols that must be returned in such an iterator and they are all stored separately in the symbol table: normal symbols, constructors, and using directives. These need to be kept separate so that they can be considered separately during lookup.

So, we add a new LinkedList that all symbols will be added to in addition to their standard storage structures. The IContainerSymbol will contain the following:

```
public Iterator getContentsIterator();
```

Originally, using directives were simply stored as a list of the nominated namespaces. However, in order for the using directives returned by the contents iterator to not be confused with normal namespace definitions, we need to store the using directives in a

different format which will allow attaching a using directive AST node to it. The following interfaces will be declared:

```
public interface IExtensibleSymbol{
   public ParserSymbolTable getSymbolTable();
   public ISymbolASTExtension getASTExtension();
   public void setASTExtension( ISymbolASTExtension obj );
}

public interface IUsingDirectiveSymbol extends IExtensibleSymbol{
   public IContainerSymbol getNamespace();
}
```

The `ISymbol` interface will be modified to extend `IExtensibleSymbol`. All interface functions dealing with using directives will be modified to use the new `IUsingDirectiveSymbol` interface.

Concerns:

- Local code blocks are only hooked into the symbol table in one direction, so while we can traverse scopes starting at the local scopes and going out, we do not see them when going in the other direction and so they will not be returned by the iterator unless they are added to the symbol table in the normal fashion.
- When enumerators are added to an enumeration, they are actually added to the scope containing the enumeration. (Fields added to anonymous structures & unions will be handled similarly). Special consideration will need to be given to this situation if we want the iterator to return the enumerators as members of the enumeration and not the containing scope.

Prefix Lookup

The symbol table must support prefix lookup for content assist. The following function will be added to the `IContainerSymbol` interface:

```
public List prefixLookup( TypeFilter filter,
                          String prefix,
                          boolean qualified )
                         throws ParserSymbolTableException;
```

The `TypeFilter` class will define what kinds of symbols should be found by the lookup. Elaborated lookup will be modified to use this class as well.

The symbol table will define a lookup mode, with two possible values: `PREFIX`, and `NORMAL`. When the lookup mode is PREFIX, then the lookup will match symbols that

begin with the given prefix. Additionally, in prefix mode, lookup will not stop once a symbol is found, and ambiguity exceptions will not be thrown but noted and the results filtered accordingly.

Content assist will require the results of this lookup to be filtered according to access visibility. Accordingly `IContainerSymbol` will have the following function:

```
public boolean isVisible( ISymbol symbol,
                          IContainerSymbol qualifyingSymbol );
```

This method will consider C++ access visibility rules to return whether or not the given symbol is visible. In order to properly do this, the symbol table will now need to keep track of friendship so the following functions will be added to the `IDerivableContainerSymbol` interface:

```
public void addFriend( ISymbol symbol )
                                throws ParserSymbolTableException;
public ISymbol lookupForFriendship( String name )
                                throws ParserSymbolTableException;
public IParameterizedSymbol lookupFunctionForFriendship(
                                     String name,
                                     List parameters )
                                throws ParserSymbolTableException;
public List getFriends();
```

Templates

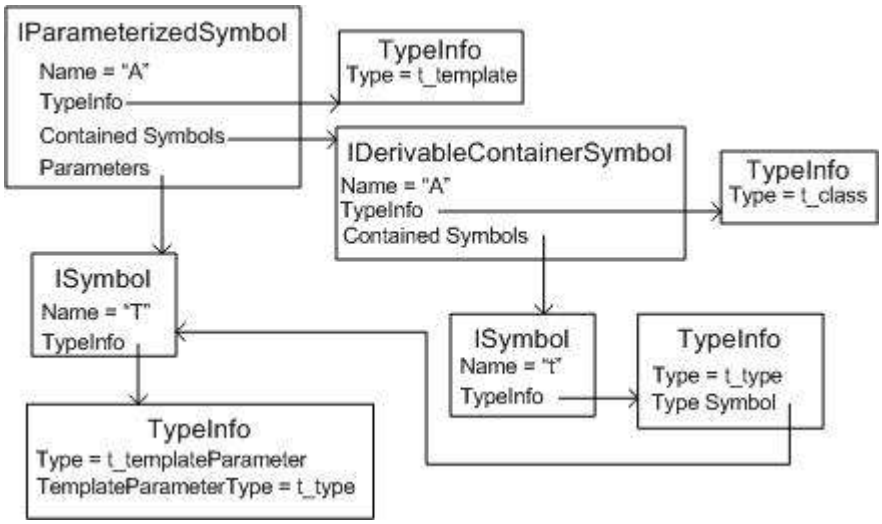
Building the Table

Templates will be represented in the table using either an `IParameterizedSymbol` or a new `ITemplateSymbol` which derives from `IParameterizedSymbol`. In either case, the template will contain the template parameters and one symbol which will be the class or function which are represented in the normal way. References to the template parameters will be handled in the normal manner as if the template parameter was a type.

Eg: The following code:

```
template < class T > A {
   T t;
};
```

would be represented as follows:

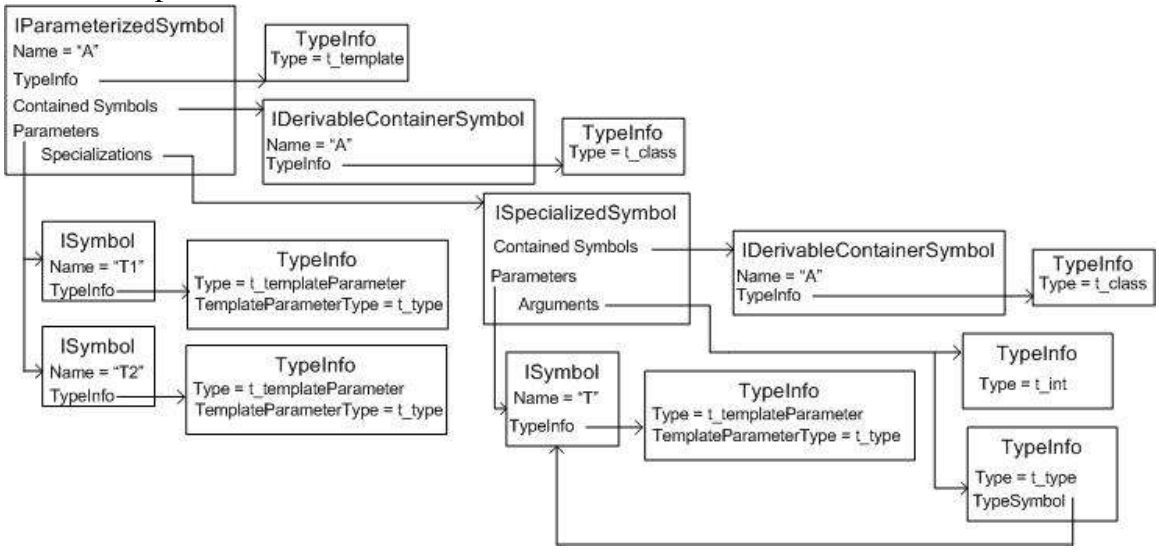


Template specializations will be represented with an `ISpecializedSymbol` which will derive from the interface used to represent templates, and will in fact be templates in their own right. `ISpecializedSymbol`s will contain both a list of parameters and a list of arguments. Each template will contain a list of all its specializations, which includes both explicit specializations and class template partial specializations.

Eg: The following code:

```
template< class T1, class T2 >  class A {};
template< class T > class A < int, T > {};
```

would be represented as follows:



When a templated symbol is looked up, the symbol table will decide based on the arguments given whether the primary template or one of the specializations is returned.

Definitions

A member function, a member class, a static data member or a member template of a class template can be defined outside the class template definition. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition.

Eg:

```
template < class T1, class T2 > class A {
   void f();
};
template < class U, class V > void A< U, V >::f(){}
```

A special lookup function, `lookupTemplateForDefinition`, will be required to lookup the template using the alternately named template parameters. Once the correct template is determined, the member definition can be hooked up to the declaration using the same method used for normal forward declarations.

Because the member definition's template parameters have different names than the template's declaration's parameters, the new parameter names must be kept so that, for example, name lookups for "U" for example inside `f()` will correctly resolve to the proper template parameter. The template can keep a map of member definition to alternate parameter list for this purpose. Because of this, it may be necessary for `lookupTemplateForDefinition` to return a factory capable of creating the member definition instead of simply returning the template symbol.

Members of class template partial specializations are unrelated to the members of the primary template. So specialization definitions can be handled in the same manner, the only difference being the selection of the proper specialization by the `lookupTemplateForDefinition` function.

Instantiation

Before a template can be referenced in code, it must be instantiated. There are two ways a template can be instantiated: implicitly and explicitly.

```
template < class T > void f( T t ) { }

template void f<int>( int t );  //explicit instantiation

f<int>( 1 );   //implicit instantiation
f( 1 );        //also implicit instantiation
```

The mechanics of both types of instantiations will be the same, though an explicit instantiation would need to be requested by the parser, and an implicit instantiation could be triggered by the symbol table during a lookup.

To actually instantiate the template, first the symbol table would need to decide which definition of the template should be used. It does this by considering the arguments and comparing the primary template and all its specializations.

Once the template or specialization has been selected, it is instantiated by cloning the templated symbol and its contents. Then, in that clone, all references to the template parameters are replaced by references to the appropriate template argument. The clone will then be used to represent the instantiated template, and for all intents and purposes it will completely resemble a normal class or function. This means that none of the existing symbol table code needs to worry about whether or not symbols it encounters are template instances.