# An Approach for Modeling the Name Lookup Problem in the C++ Programming Language (Extended Version[*])

James F. Power[1] and Brian A. Malloy[2]

[1] Department of Computer Science, National University of Ireland, Maynooth
County Kildare, Ireland
`James.Power@may.ie`
[2] Department of Computer Science, Clemson University
Clemson, South Carolina, USA
`malloy@cs.clemson.edu`

**Abstract.** Formal grammars are well established for specifying the syntax of programming languages. However, the formal specification of programming language semantics has proven more elusive. A recent standard, the Unified Modeling Language (UML), has quickly become established as a common framework for the specification of large scale software applications. In this paper, we describe an approach for using the UML to specify aspects of the static semantics of programming languages. In particular, we describe a technique for solving the name lookup problem for the recently standardized C++ programming language. We apply our approach to C++ because a solution to the name lookup problem is required for parser construction and our solution is applicable to Java and other programming languages.

## 1 Introduction

Formal grammars are well established for specifying the syntax of programming languages. Moreover, extended Backus Naur Form (EBNF) is a popular representation for language syntax; furthermore, many tools have been developed that accept regular expressions and context free grammars as a basis for automating the early stages of compiler construction. However, the formal specification of programming language semantics has proven more elusive. No single formal technique for semantics specification has gained the wide acceptance that formal grammars have attained for syntax specification. Indeed, formal semantics specification has yet to yield convincing solutions for problems associated with modularity, scalability and automatic implementation of semantic analyzers[6].

The Unified Modeling Language (UML), has quickly become established as a common framework for the specification of large scale software applications[3, 9]. The UML is widely accepted by software developers and many UML CASE

---

[*] This is an extended version of the paper presented at the 15th ACM Symposium on Applied Computing, Villa Olmo, Como, Italy, 19-21 March, 2000.

tools are now available. In this paper, we describe an approach for using the UML to specify aspects of the static semantics of programming languages. In particular, we describe a technique for solving the *name lookup* problem for the recently standardized C++ programming language[1].

The *name lookup problem* is defined as follows: given the use of a name in a program, find the corresponding declaration of that name. We present a structural model that captures syntactic information about *where* a name occurs in a program together with semantic information about *how* the name occurs in the program. We discuss various options for placement of a lookup method that, given a program name, finds the corresponding declaration for that name; we arrive at a placement of the lookup method that properly places responsibility for lookup in the semantic hierarchy of the structural model. We include a *program processor* in the model that assembles an object that contains contextual information about a name encountered in a program. The program processor than passes the newly created object to the semantic hierarchy that performs the lookup in the enclosing scope and searches other relevant scopes if the name is not found.

We apply our approach to C++ because a solution to the name lookup problem is required for parser construction and C++ parsers are difficult to construct[2, 5, 7, 8]. There is no public domain parser currently available that accepts the language described in the C++ standard. Our use of UML to specify the semantics of C++ will provide documentation of the language to UML-knowledgeable developers. Many aspects of our solution are applicable to languages with similar constructs, for example Java.

The remainder of this paper is organized as follows. The next section provides background information about the name lookup problem including definitions of terms that we use in subsequent sections. Section 3 presents the various options that we considered in our design of the structural model and Section 4 provides object and interaction diagrams that serve to elucidate aspects of the structural model. In Section 5 we draw conclusions.


## 2   The Problem Domain

In this section we review some of the basic concepts associated with C++ programs and the name lookup problem in particular. Our presentation here is necessarily informal and incomplete; the definitive description is, of course, clause three of the standard.

A *name* in C++ is the use of an identifier to refer to any of the usual C++ entitles, such as objects, functions, arrays or classes. Typically, each name is introduced by a *declaration* before its use, and this declaration usually gives some additional information about that name, such as its type. A *definition* performs the same function as a declaration, but in addition associates a value with the name. Since a definition is a special kind of declaration, in this paper we will use the word "declaration" when we mean either.

| Just a Declaration | Definition |
|---|---|
| `extern int a;` | `int a;` |
| `int f(int);` | `int f(int i) { return i; }` |
| `class A;` | `class A { int x, y; };` |

**Fig. 1.** *Declarations vs Definitions.* This figure gives some simple examples of the differences between declarations that are and are not definitions.

The *name lookup problem* involves associating each occurrence of a name in a program with its corresponding declaration. This has the effect of identifying the kind of entity to which the name refers, as well as creating an implicit association between all occurrences of a name that refer to the same entity. Since overloading is a feature of C++, performing name lookup for a given occurrence may in fact return a *set* of possible declarations.

Since each name occurrence must have a corresponding declaration, name lookup performs a validation function: returning an empty set of declarations for a given usage indicates that the program is ill-formed. However, name lookup does not consider additional restrictions that are placed on the use of a name, such as accessibility and type-correctness; in C++ these checks are performed *after* name lookup has taken place.

The *scope* of a name is that portion of the program within which it is valid to refer to that name. A language such as K&R C [4] has a relatively simple concept of scope, with each declaration being either at global scope, or at local scope inside a function. C++ enhances this considerably by providing classes and namespaces, and by allowing these various forms of scope to be nested inside each other, subject to certain restrictions.

A detailed description of C++ classes and namespaces is beyond the scope of this paper. Briefly, a *namespace* is a modularization construct that allows the programmer to *name* the region within which a series of declarations occur, and to control the visibility of these declarations through the use of this name. Classes provide the object-oriented aspects of the language, implementing encapsulation through various forms of access restriction, and re-use through inheritance.

A class may be enclosed by another class, a function or a namespace. A class directly enclosed by another class is called a *nested class*, whereas a class inside a function is called a *local class*. A function may be directly enclosed by either a class or a namespace. A namespace may only be enclosed by another namespace. Global scope encloses all other scopes, and may be regarded as a special instance of a namespace scope.

## 3   The Evolution of the Structural Model

In this section we describe the design options that we considered in our approach to solving the name lookup problem. Clause three of the recently adopted C++ standard enumerates the procedures for accomplishing name lookup; this enumeration forms the use cases that drive our model. We present two different
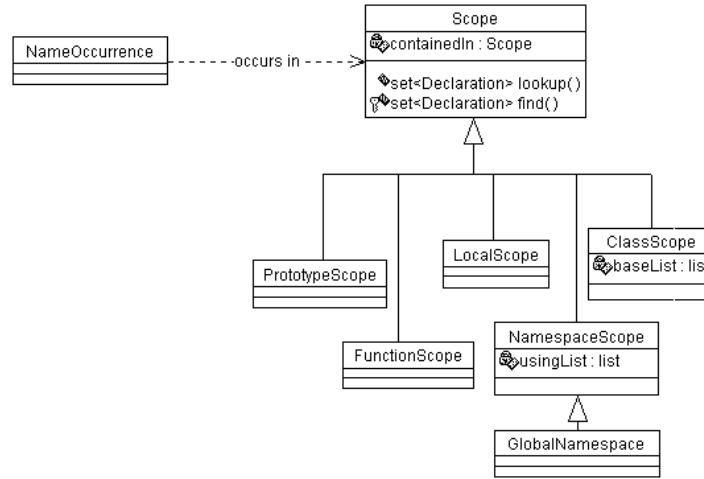
**Fig. 2.** *Class Diagram representing the scope hierarchy.* This figure enumerates the different scoping constructs in the C++ language and the relationships between them.

approaches to the construction of a class diagram that realizes the use cases in clause three and captures the static semantics of C++ with an explanation of the advantages and disadvantages of each approach. In the next section we overview the important classes in each approach and Section 3.2 follows with a presentation of an approach that places the burden of lookup on the syntax of the language. Finally, Section 3.3 presents an approach that places the burden of lookup on the semantics of the language.

### 3.1   Overview of the Approach

The class diagram in Figure 2 illustrates the important classes in the structural model that we use to solve the name lookup problem, and the associations between the classes. The class on the left side of the Figure, NameOccurrence, represents the physical occurrence of a name in the program text, where the occurrence can be a declaration, definition or any usage of a name. Each such occurrence of a name occurs in the context of a logically enclosing scope, which is represented by the hierarchy on the right side of the figure. For the name lookup problem, a scope is a list of names declared and therefore available in a logical region of the program. As part of a typical implementation of a program parser, these scopes are usually represented by a *symbol table*. Further consideration of the representation of a name in a symbol table is not important for the problem that we consider here. We include two methods, find, a private method that simply searches through those names defined in the current scope and lookup

that drives the overall search procedure in the context of the current scope, and provides a public interface to find. Since every scope other than global scope may be logically nested within an enclosing scope, we provide a containedIn attribute to capture this relationship; the containedIn attribute is shown as a private data member of class Scope in Figure 2.

The C++ standard distinguishes five kinds of scope that we choose to represent through subtyping. Thus, base class Scope in Figure 2 has five subclasses, LocalScope, ClassScope, NamespaceScope, FunctionScope and PrototypeScope. The first three of these classes correspond to local scope in a block of the program, class scope and namespace scope. Both ClassScope and NamespaceScope have specialized mechanisms for referring to other scopes that we represent using attributes: a list of base classes for ClassScope and a list of imported namespaces in namespace scope represented by baseList and usingList respectively. Every program contains an implicit namespace for global attributes; we represent this implicit namespace with the singleton class GlobalNamespace derived from NamespaceScope. Finally, classes PrototypeScope and FunctionScope, also derived from Scope, represent two less familiar scope levels. Function prototype scope refers to the fact that parameters in a function declaration are in scope for the duration of that declaration. Function scope refers specifically to the use of label declarations and goto statements within a function.

In summary, the NameOccurrence class captures local information about *where* a name occurs in the program; this class represents a syntactic view of the program that incorporates contextual information into the model. The Scope hierarchy captures global information about *how* a name occurs in the program; this hierarchy represents a semantic view of the program that incorporates information about the organization of the scopes.

Since the goal of name lookup is to associate a name in the program with its corresponding declaration, an important consideration is where in the model is the burden of lookup placed. In the sections that follow, we explore two options for the placement of lookup.

### 3.2   A Context-Centered Approach to Lookup

Our first approach to modeling name lookup adopts a context-centered view of the problem that places a lookup method in class NameOccurrence. In clause three of the recently adopted C++ standard, the name lookup problem is clearly defined including a detailed enumeration of the procedures for accomplishing lookup. The procedural enumeration in clause three is organized by context where the list of contexts include (1) an unqualified name, (2) an argument-dependent name, (3) a qualified name, (4) a name that includes an elaborated type specifier, (5) a class member name, or (6) a name that's a target of a using directive or namespace alias.

Figure 3 illustrates the incorporation of context into the structural model by subtyping the class Usage with a class for each of the contexts listed in the C++ standard. Figure 3 only lists classes ClassQualified and NamespaceQualified as examples of further subtyping; if this approach were adopted the subclasses
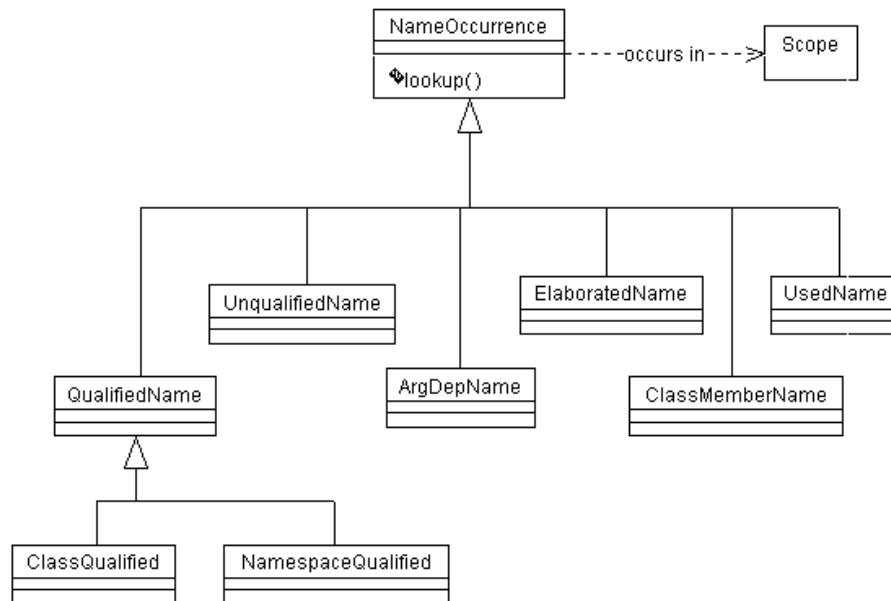
**Fig. 3.** *Class Diagram for the context-centered approach.* This figure illustrates the class hierarchy required representing the context in which name lookup occurs, paralleling the structure of clause three of the C++ standard.

of NameOccurrence would themselves be subtyped by the classes enumerated in clause three of the C++ standard.

Name lookup in the context-centered approach proceeds as follows. As the program processor encounters a name in the program, the context in which the name occurs is assembled into an instance of NameOccurrence. This object encapsulates the logic of the search procedure as specified in the C++ standard and represented by the lookup method in NameOccurrence. The lookup algorithm choreograph the search by consulting with the current scope object, ordering a find and, if unsuccessful, querying the scope to determine its logical parent and subsequent target for searching.

In this context-centered approach, control of the lookup process is centered in the NameOccurrence hierarchy and the role of the Scope hierarchy is to provide information about the names stored at a given scope and the identity of its related scope, as requested. Since the procedures for name lookup are organized by context in clause three of the C++ standard, this structural model will be easily understood with reference to the standard. In addition, verification based on the C++ standard can proceed in a case by case manner.

However, there are several problems in the context-centered approach. The first problem is that partitioning control based on the structure of clause three
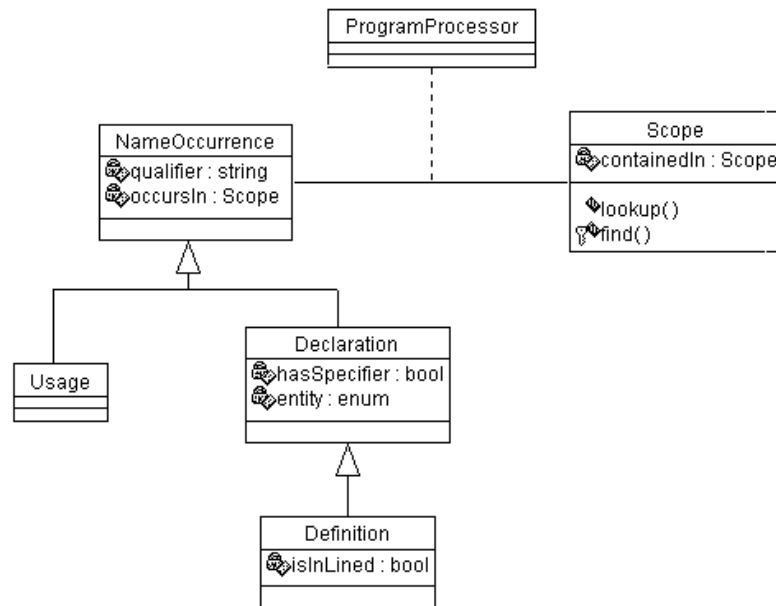
**Fig. 4.** *Class diagram for the semantic-centered approach.* This figure illustrates how the diffusion of control to the Scope and ProgramProcessor classes allows us to considerably simplify the NameOccurrence hierarchy. The ProgramProcessor class, illustrated at the top of the figure, establishes the relationship between NameOccurrence and Scope classes by passing the former as a parameter to the lookup method of the latter.

of the C++ standard induces methods that manifest a high degree of coupling between classes. This coupling results from the fact that the single instance of a name occurrence can incorporate features from many other classes. For example a lookup of member name m in o.Q::m(a) combines instances of QualifiedName based on qualifier Q, ArgDepName based on argument a and ClassMemberName for the object o.

A second problem with this approach is that once a lookup is initiated from a particular object in the NameOccurrence hierarchy, the logic of control proceeds similarly for all such objects since it effectively mirrors the structure of the Scope hierarchy. By restricting the context to initiation of control and permitting the Scope object to complete the search process we avoid the duplication of search logic in each of the NameOccurrence objects and this reorganization permits coalescing and considerable simplification of the NameOccurrence hierarchy.

Furthermore, once the context of an occurrence has been resolved and control passed to the relevant Scope object, the object can either complete the lookup itself or it can nominate a Scope object to continue the search until the lookup is complete. This process of initially determining syntactic context and then pass-

ing control to semantic processing closely parallels the actions of a processor based on an attribute-grammar that selects an appropriate grammar rule, determines syntactic context, and then performs the corresponding semantic action.

### 3.3   A Semantic-Centered Approach to Lookup

Figure 4 illustrates an alternative, semantic-centered approach to solving the name lookup problem. Here, the lookup function that controls the search strategy has been transferred to class Scope. The NameOccurrence class and its subclasses are now information carriers and do not encapsulate any search capabilities of their own. That part of the search strategy concerned with assembling context information is now delegated to a new class ProgramProcessor, which we may regard as an abstraction of the parser.

The class hierarchy represented by NameOccurrence, has been reorganized based around the kind of context information required by the lookup method of the Scope class. We are now in a position to model the categorization of name occurrences in terms of declarations, definitions and uses as described early in clause three of the C++ standard. The distinction between qualified and unqualified names is now represented by an attribute in the NameOccurrence class while more specific information, such as the presence of an elaborated type specifier, is represented as an attribute of the Declaration class.

As the ProgramProcessor encounters a name, since it knows the context of the name from the grammar, it assembles this information in an instance of the NameOccurrence class and then passes this object, through the lookup method, to the enclosing scope. The enclosing scope can now use the context incorporated in the object to make suitable decisions during the lookup process, e.g., ignoring enclosing namespaces during a qualified name lookup.

## 4   Behavioral Modeling of Name Lookup

While the class diagrams of the previous section provide a general framework within which name lookup takes place, we still have not specified the actual rules used in deciding the sequence of scopes that are searched, and which would be encoded in the `lookup` method for these scopes.

As might be expected with such a complex language as C++, there are many cases to consider, and indeed, several examples are presented in the C++ standard. In this section we present two specific cases, namespace lookup and class-based lookup, which capture many of the fundamental decisions involved in the lookup process.

The examples presented below are based on some of those presented in the C++ standard. These examples typically present a series of declarations or definitions, and then some usages, and then describe the name lookup for these usages. We observe here that the standard UML modeling techniques of object diagrams and sequence diagrams fit smoothly onto this pattern, since the former are ideal for describing the environment created by a declaration sequence, and

```
 1  void f(float);
 2  namespace Y {
 3    void f(char);
 4    void h(double);
 5  }
 6  namespace Z {
 7    void f(float);
 8    void h(int);
 9    namespace A {
10      using namespace Y;
11      int i;
12    }
13  }
14  namespace B {
15    using namespace Z;
16    void f(int);
17    void g(int);
18  }
19  namespace AB {
20    using namespace Z::A;
21    using namespace B;
22    void g();
23  }
24  void h() {
25    AB::f(0);
26  }
```

**Fig. 5.** *Namespace example.* This program segment is used to demonstrate name lookup for namespaces. We search a series of nested namespaces for the declaration corresponding to name **f**.

the latter are ideal for describing a particular invocation of the name lookup procedure.

### 4.1  Name Lookup for Namespaces

Namespaces act as a modularization construct in C++, allowing the programmer to partition the names used in a program to prevent them from interfering with each other. Thus, given some variable x declared in namespace A, once outside namespace A we may refer to the variable using explicit qualification, as in A::x.

Namespaces may be nested inside each other, in which case name occurrences inside the inner namespace may refer to those already declared at the outer level without the need for qualification with this process continuing recursively, eventually reaching the global namespace where all namespaces are ultimately nested and this is a run-on sentence that James and I must fix.

In addition to this textual relationship achieved through qualification or nesting, we may establish a logical relationship between namespaces by importing
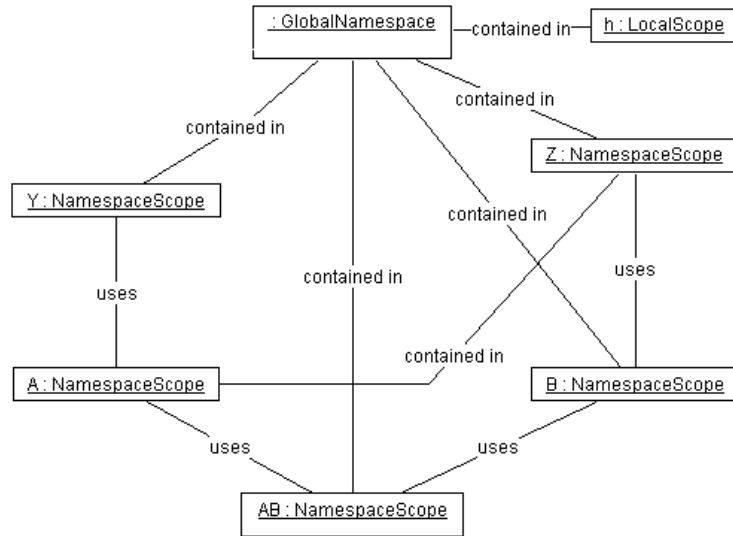
**Fig. 6.** *Object diagram for the namespace example.* This figure represents the two kinds of relationships between namespaces: those established by a using directive and those established by nesting.

one into another with a `using` directive. The declarations in a namespace that are imported in this way are treated as though they were originally declared in the importing namespace.

In Figure 5 we list a program segment to illustrate name lookup for namespaces. Here we declare five namespaces, with various using relationships and one nesting relationship; we also define a single function `h()`.

Our first step in representing this example is to construct an object diagram, illustrated in Figure 6, showing how each scope instantiates a corresponding subclass of our model's **Scope** class, illustrated in Figure 2 and discussed in Section 3.1. The object diagram also represents the nesting and import relationships between the namespaces. For example, all namespaces are contained in an implicitly declared global namespace; thus, namespaces Y, AB, B and Z of Figure 5 are implicitly contained in an unnamed global namespace. This nesting relationship is captured in Figure 6 by the unnamed class at the top of the figure that is an instance of **GlobalNamespace**, and classes Y, AB, B and Z related to the unnamed instance of **GlobalNamespace** by the contained in connector. Similarly, namespaces A and B of Figure 5 contain named using directives for Y and Z respectively. This import relationship is captured in Figure 6 by the uses connector between instances of Y and A, and between Z and B. Similar uses relationships are shown in Figure 6 between A and AB and between B and AB.
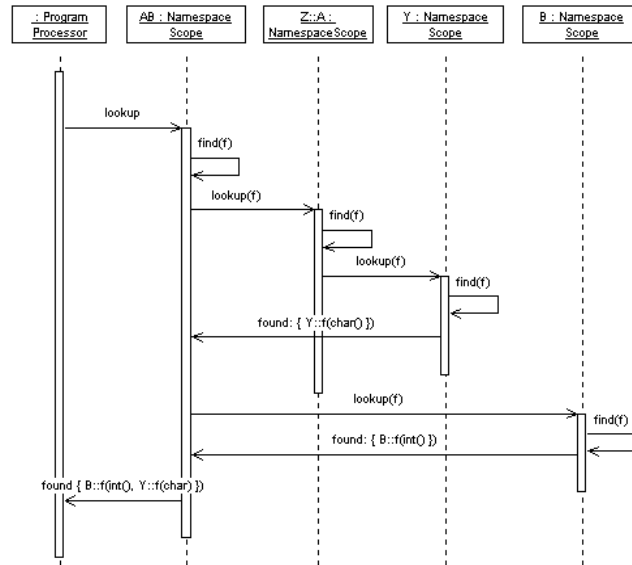
**Fig. 7.** *Sequence diagram for the namespace example.* This figure captures the flow of control as messages are passed through the namespace hierarchy.

At the end of the program segment illustrated in Figure 5 we have function `h()` at global scope, which contains a usage of the name `f`, with an explicit qualification by the namespace `AB`. To illustrate name lookup for `f`, we present a sequence diagram, illustrated in Figure 7, indicating the scopes that are searched and the order that they are searched. The lookup proceeds as follows:

- Namespace `AB` is first searched unsuccessfully; we must next search its imported namespaces `Z::A` and then `B`.
- The search of namespace `Z::A` yields no declaration, and so we search its imported namespace `Y`. Our search of `Y` is successful.
- Going back to `AB` we are now directed to search namespace `B` where we also find a declaration, thus precluding the search of its imported namespace `Z`.

The search terminates, returning the two possible definitions of `f` to the program-processor, which will then proceed with the subsequent stages of processing such as overload resolution.

Note that we do not consider the namespace `Z` in which `A` is nested in this part of the search, nor do we search global namespace, in accordance with the lookup rules for qualified names. The presence of a qualifier is made known to the Scope's `lookup` method through its parameter, the NameOccurrence instance corresponding to `f`.

In the sequence diagram we are able to explicitly represent the roles and responsibilities of each scope level. For example, it is the namespace `AB` that

```
1  namespace M {
2    int i;
3    class B { };
4  }

5  namespace N {
6    class Y : public M::B {
7      friend void foo() ;
8      void fun() {
9        class X {
10         int a[i];
11       };
12     }
13   };
14 }

15 namespace N {
16   void foo() {
17           i = 5;
18   }
19 }
```

**Fig. 8.** *Classes example.* This program segment is used to demonstrate name lookup for classes. We search a series of local and base classes for the declaration corresponding to the two usages of name `i`.

causes a search of `Z::A` and then `B` (in that order), and which is responsible for combining the results from each of these searches.

### 4.2   Name Lookup for Classes

In our second example we show how name lookup proceeds in the presence of class definitions. Classes in C++ may inherit from multiple base classes, and may be nested inside namespaces, inside functions and inside other classes. Ignoring access restrictions, the rules for lookup of a name used inside a class tells us to search that class and then recursively search its enclosing scopes, unless the class has base classes, in which case these will be searched *before* any enclosing scope.

The program in Figure 8 contains a relatively intricate web of relationships between the different scope levels, which we represent in the object diagram of Figure 9. Class X is a local class of method `fun()` that belongs to class Y, and class Y in turn has a base class B, declared in a different namespace.

In our first example of name lookup for classes, presented in Figure 8, we search for a declaration to match the usage of variable `i` on line 10. Since this variable is used outside a method body, we first search the enclosing instance ClassScope for class X. We then proceed through the immediate enclosing scopes:
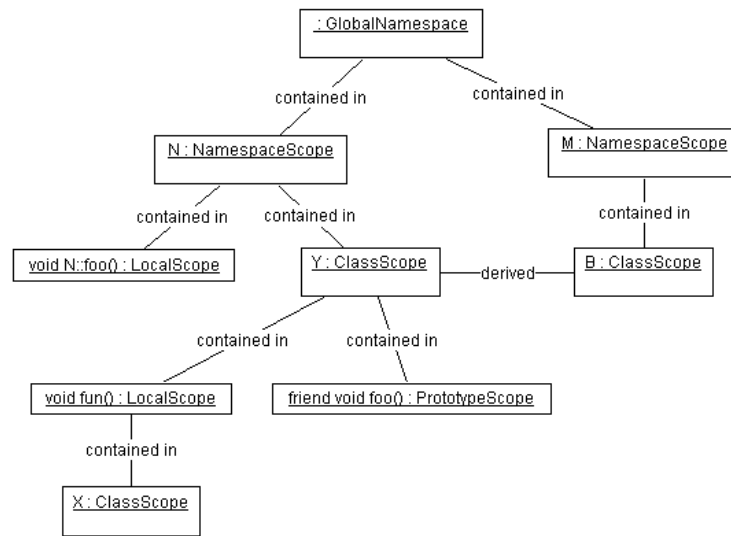
**Fig. 9.** *Object Diagram for the classes example.* This figure represents the containment relationship between the functions, classes and namespaces used in the sample program and also a derivation relationship between class Y and its base class B.

the member function `fun()`, and its enclosing class Y. Class Y is then responsible for initiating the search of its base classes (in this case class M::B), and then of its enclosing namespace N. Finally, the name space enclosing N, `GlobalNamespace`, is searched.

The second example, presented in Figure 8, shows how name lookup proceeds for the usage of i on line 17 of the program. Here the occurrence is in the body of one of class Y's friend functions which, since it is not declared inline, is treated as though it were an ordinary member of the namespace in which it is defined. Thus we carry out a standard unqualified search in the local scope corresponding to function `foo()`, in namespace N, and then its enclosing namespace GlobalNamespace.

In both cases the contents of namespace M other than class B are ignored, and no corresponding declaration for variable i is found.

Since the usage of i on line 10 occurs outside a member function, we search only that part of class X declared before the usage. This contrasts with a usage inside a member function, where the whole of the member's class is available.[1] The `lookup` function here can determine which type of search is appropriate by checking the context information stored in the Usage information passed to it.

---

[1] Such a search would have to be implemented by a backpatching procedure that would be the responsibility of the program-processor, whose details we omit.
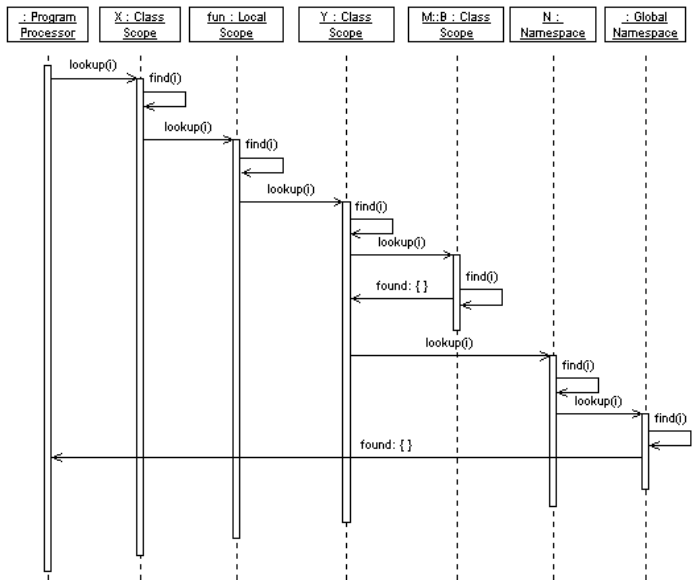
**Fig. 10.** *Sequence Diagram for the first example of name lookup in classes.* In this figure, we search for the declaration that corresponds to the usage of name i that occurs inside class X.

## 5    Concluding Remarks

In this paper we have presented an approach for modeling the name lookup problem for the C++ programming language. We have presented a structural model that captures the syntactic and semantic aspects of name occurrence in two separate but related class hierarchies. We have included interaction diagrams that elucidate aspects of the structural model by describing examples of name lookup closely modeled on those found in the C++ standard. We presented two possible views of the structural model corresponding to a context-centered approach and a semantic-centered approach to the solution of the name lookup problem. The semantic-centered approach provides advantages over the context-centered approach that was based on the structure described in the C++ standard. The C++ standard is both a rule-based document and the product of many years of C++ compiler implementations, and we may assume that the experiences gained from the latter process contributed to the coherence of the context-centered approach.

## References

1. ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, first edition, September 1998.
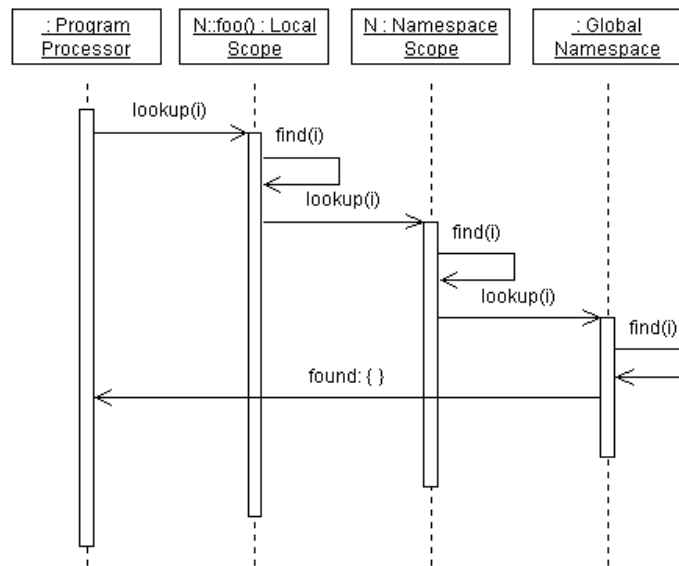
**Fig. 11.** *Sequence Diagram for the second example of name lookup in classes.* In this figure, we search for the declaration that corresponds to the usage of name i that occurs inside the friend function N::foo().

2. F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *OON-SKI*, pages 122–136, Oregon, USA, 1994.

3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

4. B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

5. John Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.

6. P.D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 11, pages 575–631. Elsevier, 1990.

7. S.P. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.

8. J.A. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1989.

9. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc, 1999.

# 6    Appendix

To further demonstrate the credibility of our solution to the name lookup problem, this section presents solutions to the problem for examples found in clause three of the recently adopted C++ standard[1]. Moreover, this section presents solutions for all examples in this clause except for those that are trivial or are similar to those examples found in Section 4 of this paper.

It is not intended that this appendix should either act as a substitute for, or be read independently of, clause three of the standard. We have deliberately avoided repeating much of the explanatory information in clause three, and our comments here are limited to those issues of relevance to a specific diagram. While the object and sequence diagrams presented here may be seen as an illustration of the examples, it should be noted that their construction played a vital role in both the design and validation of the class diagram.

The information presented here is not an exact mirror of that presented in the examples. Specifically:

- The object diagrams are an abstraction of the structure found in the examples, since they omit some of the declarations and statements, and since they reflect the logical, rather than the physical, relationships between the various scopes. For example, we can present the existence of a `using` relationship between namespace scopes, but not the order in which these directives occurred.

- The sequence diagrams often present more information that that contained in the examples, since they show each step of the name lookup process. While this is an advantage in terms of explicating these examples, it does on occasion force us to commit to decisions which might more properly be left to a given implementation. For example, in situations where more than one scope is searched, the standard often does not specify the exact sequence in which this must be done; however, preparing the diagrams forced us to make a specific choice.

We organise this appendix by the sub-clauses that appear in the C++ standard and the sections that follow parallel these sub-clauses. In particular, for each example of the C++ standard we present an object diagram showing the relationships between the scopes used in the example, and a sequence diagram for each name lookup discussed in the example. We annotate each diagram with the section and paragraph of the corresponding example in clause three: for example, "§3.4.1/3" refers to section 3.4.1, paragraph 3.

The examples in this section use the class diagrams found in Section 3; in particular, the interested reader may refer to Figures 2 and 4.

## 3.4.1 Unqualified name lookup

### Paragraph 3

This first example from section 3.4 involves a particularly intricate set of decisions, many of which are tangential to the name lookup problem. The relationships between the scoping levels is easily represented in the object diagram:
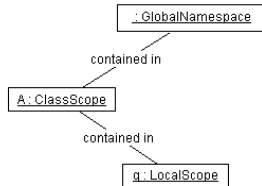


*§3.4.1/3: Object diagram.*

We have omitted the function-prototype scope corresponding to the declaration of the friend function here as it plays no part in the lookup process. In future examples we will omit irrelevant function prototype scopes in order to prevent the object diagram becoming cluttered.

Since a friend declaration doesn't introduce a new name into the scope of the class (section 11.4/1), the lookup for f finds the typedef at namespace scope. This causes the expression to be interpreted as an explicit type conversion (section 5.2.3/1), and the lookup for the name a finds the corresponding formal parameter.
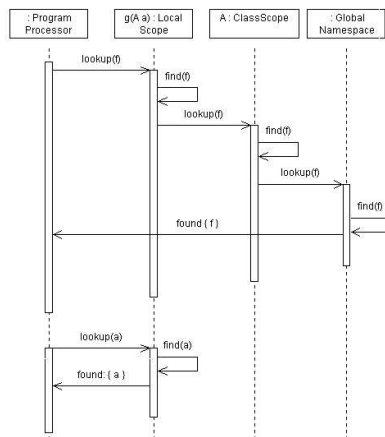


*§3.4.1/6: Sequence diagram for lookup of f(a).*

**Paragraph 6**

In this paragraph we have an example of a name used inside a function, where that function belongs to a nested namespace, but is defined outside of that namespace. The object diagram represents the nesting of the namespaces and the local scope corresponding to function `A::N::f()`.
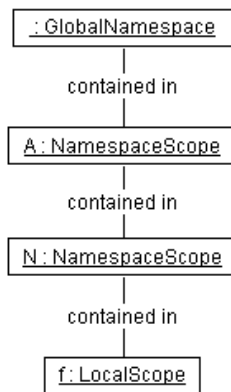


*§3.4.1/6: Object diagram.*

The sequence diagram then shows the order in which these scopes are searched, with each scope being responsible for conducting a search of itself and, if this is unsuccessful, passing control to its logically enclosing scope.
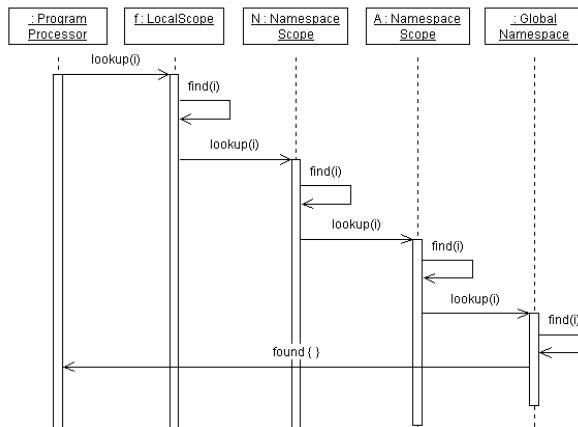


*§3.4.1/6: Sequence diagram for lookup of* `i`.

We note here that our concept of scope is *dynamic*, in that we envisage the `find` method returning different results at different stages of program processing. In particular, the `GlobalNamespace` searched here will contain all those global variables declared before the definition of function `A::N::f()`, including any that might have been declared since the end of the definition of namespace `A`.

**Paragraph 7**

Here we have an example of lookup for a name occurring inside a class definition, but outside a member function definition. The object diagram represents the logical containments between the various scopes, as well as the derivation of class Y from its base class M::B.
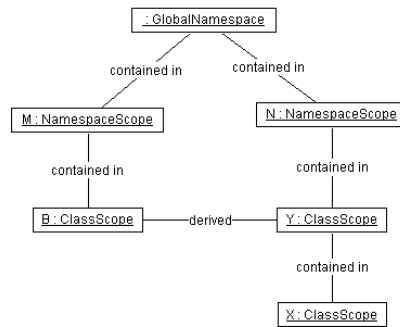


§*3.4.1/7: Object diagram.*

Derivation is represented by the **derived** relationship of the object diagram. We assume here that this means "directly derived from", and that we do not choose to represent the transitivity of class derivation.

Name lookup then proceeds in a manner roughly parallelling these logical containments. Note that the **ClassScope** `Y` is responsible for initiating two calls to the `lookup` method: one to it enclosing namespace `N`, and one to its base class `M::B`.
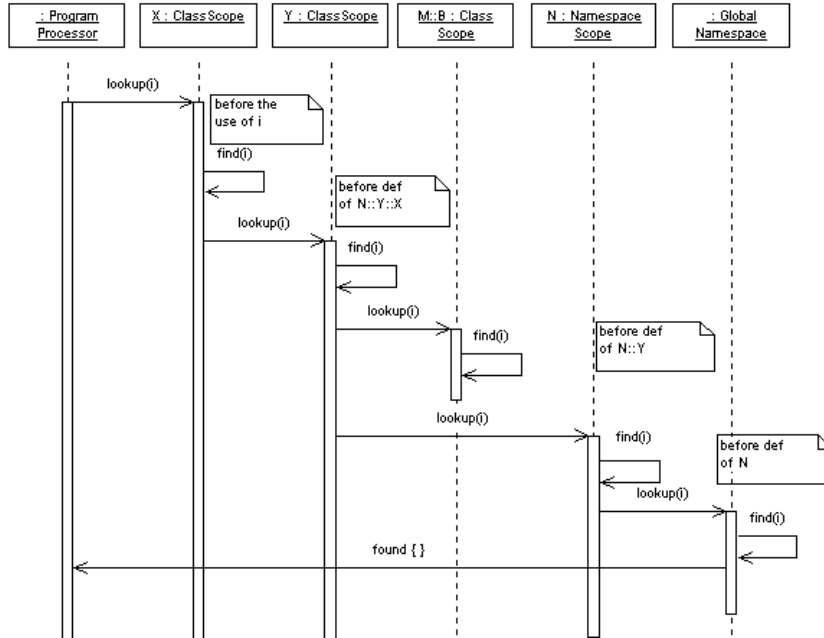


*§3.4.1/7: Sequence diagram for lookup of* `i`.

**Paragraph 8**

This example shows the lookup for a name that occurs inside the definition of a class member. As with the previous example, most of the relationships between the scopes are logical containment, with just one other relationship representing the derivation of class X from class B.
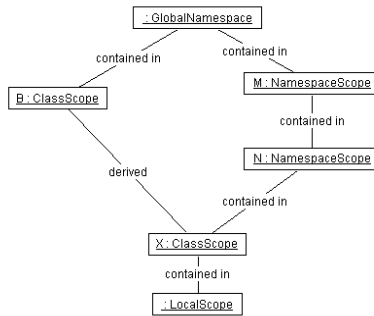


*§3.4.1/8: Object diagram.*

One again the ClassScope X is responsible for initiating a lookup of both its base class B and its enclosing namespace M::N.
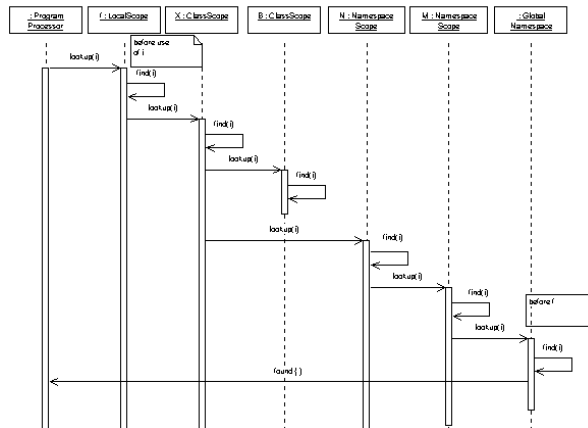


*§3.4.1/8: Sequence diagram for lookup of* i.

While the issue does not arise here, special arrangements will clearly be required in order to deal with the use of names inside a member that is defined inline, since all members of the owning class, and not just those declared to date, will be in scope. While it is the responsibility of the program processor to arrange the appropriate mechanisms for this (e.g. some form of backpatching), we envisage that the information regarding whether all of the class should be searched, or just the class-to-date, would be part of the context encapsulated with i and passed via the calls to lookup.

**Paragraph 10**

Here we have a special case of a friend function definition where the logically-enclosing scope of that function's definition is searched for names used in the declarator part of its declaration. (This is not to be confused with argument-dependent name lookup, where somewhat the reverse happens).
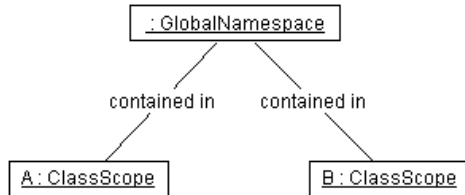


§*3.4.1/10: Object diagram.*

The sequence diagram demonstrates that the lookup message is initially passed directly to the owning scope for each friend, and only if this fails is it then passed to the physically enclosing scope, which in this case is the **ClassScope** corresponding to struct B.
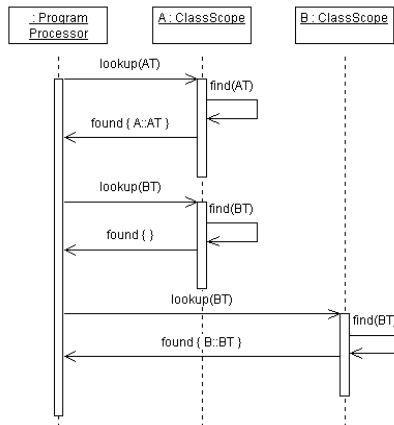


§*3.4.1/10: Sequence diagram for lookup of* AT *and* BT.

### 3.4.2 Argument dependent name lookup

**Paragraph 2**

The example in this paragraph demonstrates how the arguments in a function call extend the set of scopes that must be considered when searching for the corresponding function definition.

We choose to include a function-prototype scope for the declaration of f in the object diagram to indicate its presence in namespace NS: the actual scope plays no part in the lookup.
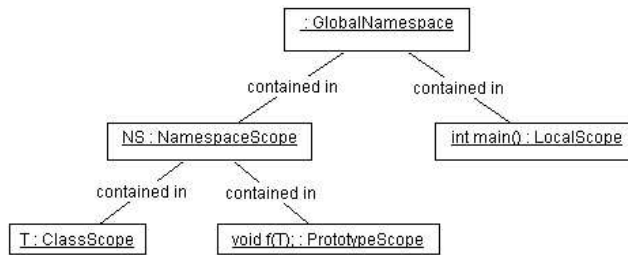


*§3.4.2/2: Object Diagram.*

The sequence diagram shows the argument parm being resolved first; this allows the program processor to assemble the set of associated namespaces and classes, which it can then use to direct the search for f.
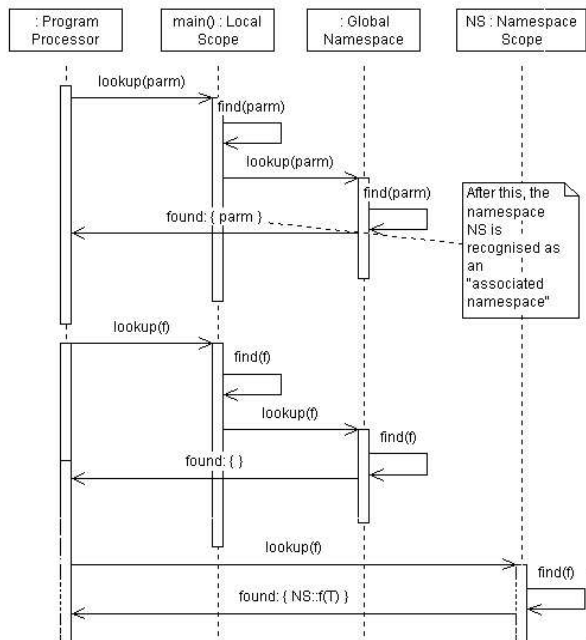


*§3.4.2/2: Sequence diagram for lookup of* f(param).

### 3.4.3 Qualified name lookup

**Paragraph 1**

This example demonstrates that the normal rules for choosing between different declarations of the same name do not apply when that name is used as a qualifier; instead we are looking for namespace or class names.
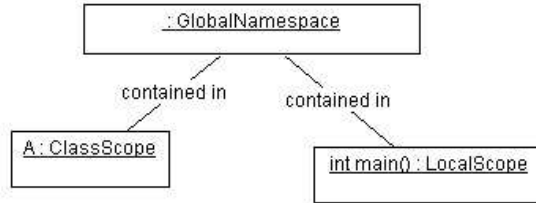


*§3.4.3/1: Object diagram.*

The sequence diagram demonstrates the different lookup procedure that is carried out in the two cases: first where A is used as a qualifier, and then where it is not.
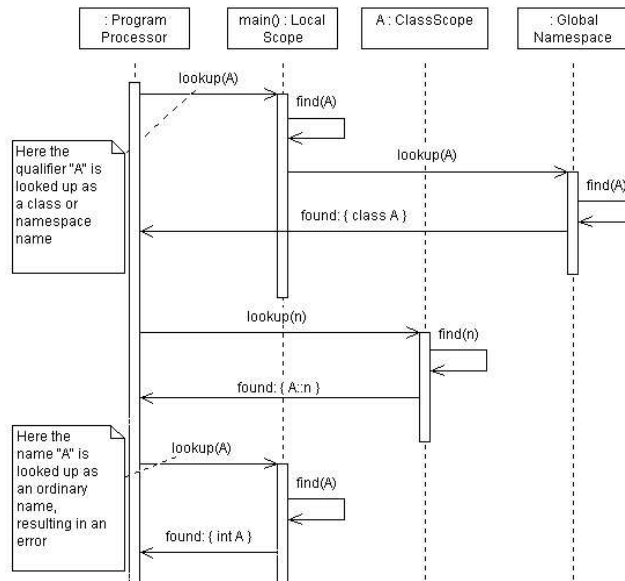


*§3.4.3/1: Sequence diagram for lookup of* `A::n`.

Note that in each case the initial method invocation appears to be the same: the program processor passes a `lookup(A)` method call to the **LocalScope** corresponding to function `main()`. The crucial difference here is in the parameter `A`, which must contain information as to whether or not this name has been used as a qualifier. Thus, the fact that these two lookups actually use different objects, both of which we call "`A`" is not represented explicitly in our sequence diagram.

**Paragraph 3**

This example shows that if a name being declared has an explicit qualifier, then this qualification is implicitly extended to subsequent names used in the declarator.
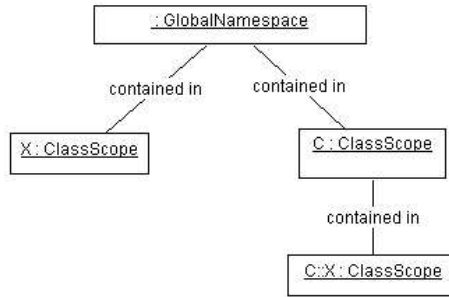


§*3.4.3/3: Object diagram.*

The sequence diagram shows that whereas the lookup for names X and C proceed as normal, as soon as we process the qualifier C, subsequent names in the declarator (i.e. arr and number) are looked up under this qualification.
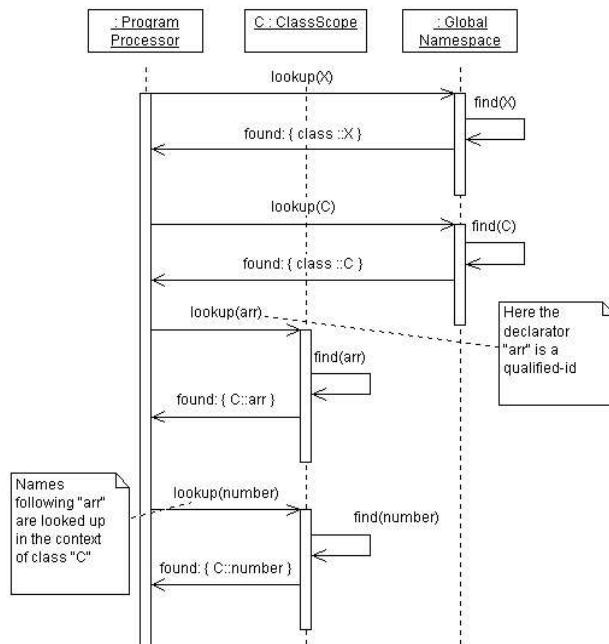


§*3.4.3/3: Sequence diagram for lookup of* X C::arr[number].

It is the responsibility of the program processor to take note of this explicit qualification, and to ensure that the objects corresponding to both arr and number are both marked as containing an explicit qualification.

**Paragraph 5**

This paragraph deals with the rules for pseudo-destructor names. In our presentation here we have one object diagram to represent the (rather trivial) scope containments, and one sequence diagram each to represent the lookups for names I, I2 and AB.
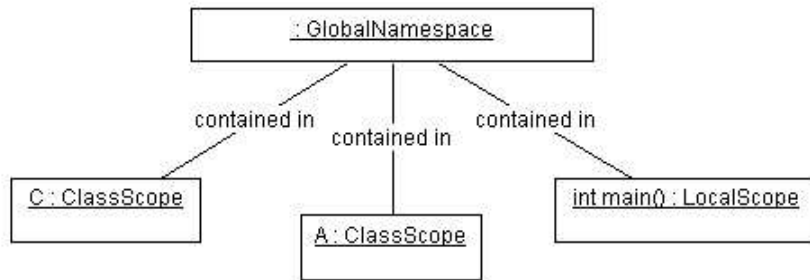


*§3.4.3/5: Object diagram.*

In the first example, the lookup of p proceeds as normal, and then the rules from section 3.4.5 cause a lookup of C in the scope of the expression, which is the GlobalNamespace. Next the two instances of the name I must be looked up in the scope designated[2] by their qualifier, which in each case is the ClassScope corresponding to struct C.

It is the responsibility of the program processor to make sure that the search for I takes place as a search for a type name, and not a search for a ("real" rather than pseudo) destructor of that name.
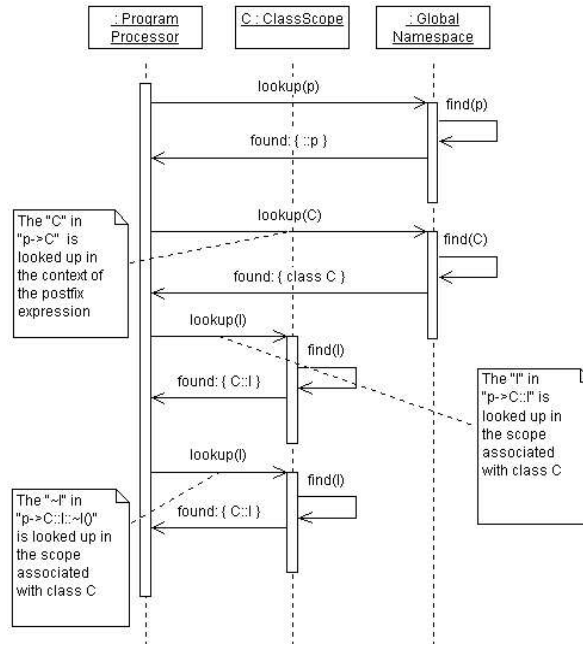


§3.4.3/5: Sequence diagram for lookup of p->C::I::~I().

---

In the second example, since q is a pointer to a scalar type, the rules from section 3.4.5/2 tell us to lookup I1 in the scope of the whole expression (i.e. the GlobalNamespace). Next, the type-name I2 is looked up in the scope designated by its qualifier I1, which is once again the GlobalNamespace.
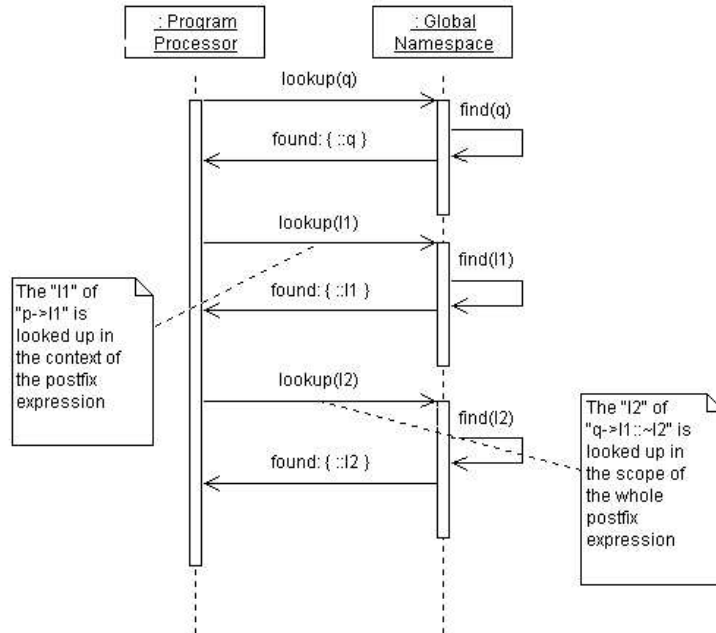


§3.4.3/5: Sequence diagram for lookup of q->I1::~I2().

In the final example, since the name AB refers to something of class-type, this is a "real", rather than a pseudo, destructor call. Hence the first AB is looked up following section 3.4.5/4 as a qualifier in both the scope of the whole expression, and in the scope of the class corresponding to p. The second AB is then looked up, not as a type-name, but as a destructor name in the ClassScope corresponding to struct A.
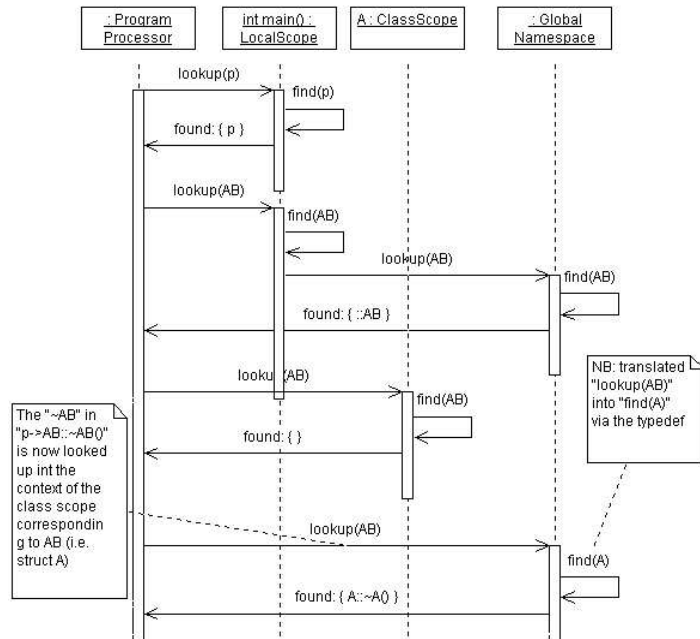


§3.4.3/5: Sequence diagram for lookup of p->AB::~AB().

### 3.4.3.2 Namespace members

### Paragraph 2

The examples in this paragraph demonstrate the searching of namespaces which are connected by a uses relationship. The object diagram represents both this relationship, and the usual logical containment.
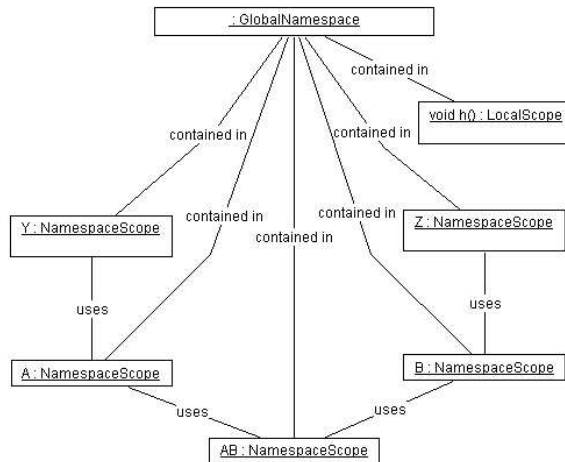


§3.4.3.2/2: Object diagram.

The four sequence diagrams that follow illustrate the lookups caused by processing the expressions in the body of function h(). In each case the variables used are explicitly qualified by the namespace AB, and hence the lookup is restricted to this namespace, and those transitively related to it via the uses relationship. Thus neither the GlobalNamespace, and the LocalScope corresponding to h() are searched in any example.

In all cases the searches are straightforward. We have chosen to make each namespace responsible for dispatching the relevant lookup calls to the namespaces it uses, and for collating the results of these calls. Note that each namespace must also be capable of choosing not to search further when a find is successful.
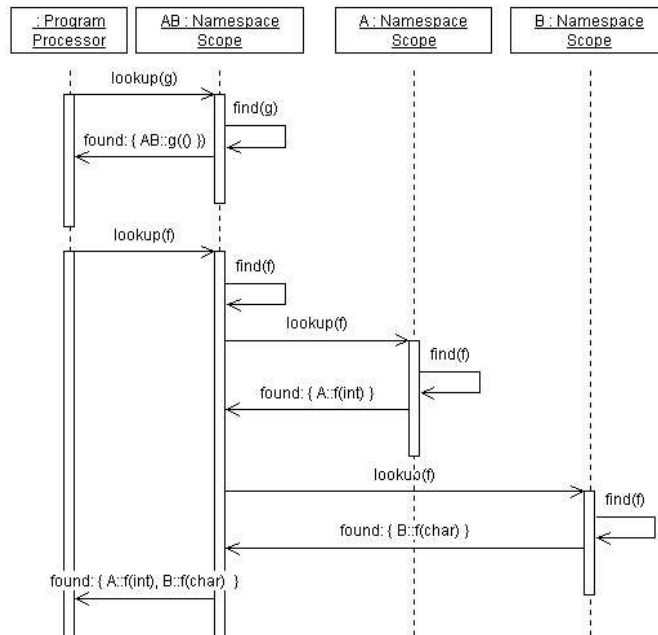
§3.4.3.2/2: Sequence diagram for lookup of AB::g() and AB::f(1).
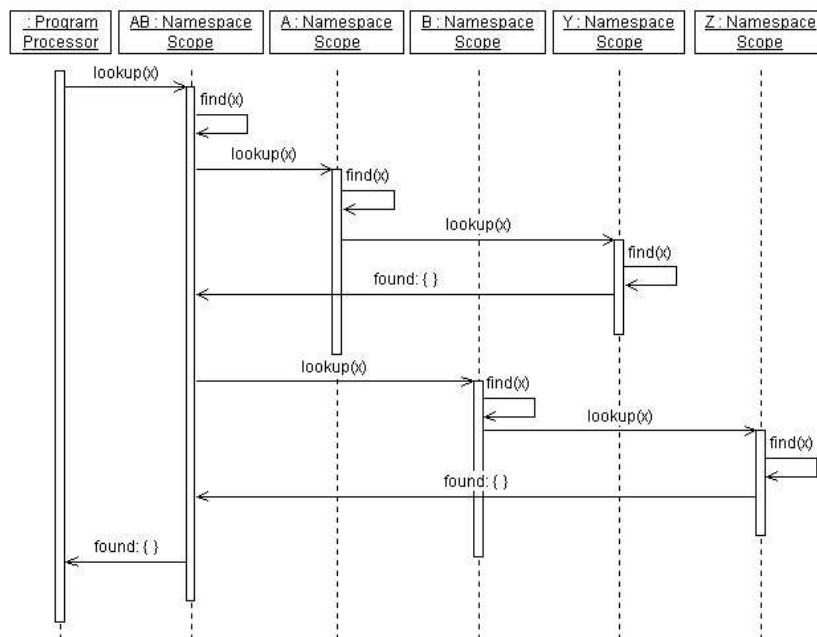


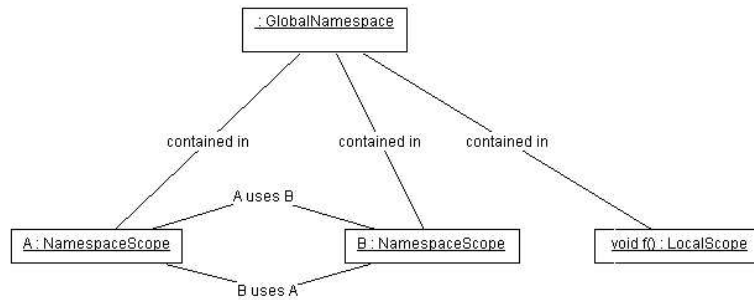§3.4.3.2/2: Sequence diagram for lookup of AB::x++.
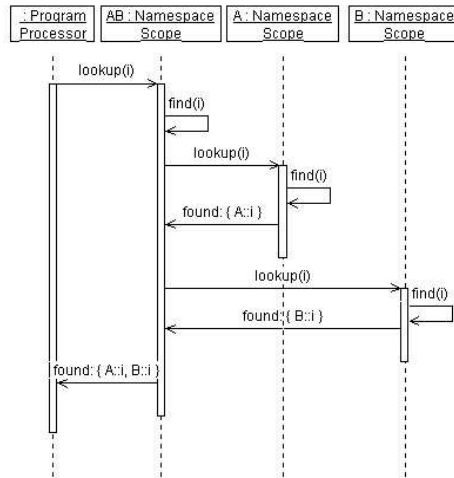
§3.4.3.2/2: Sequence diagram for lookup of `AB::i++`.
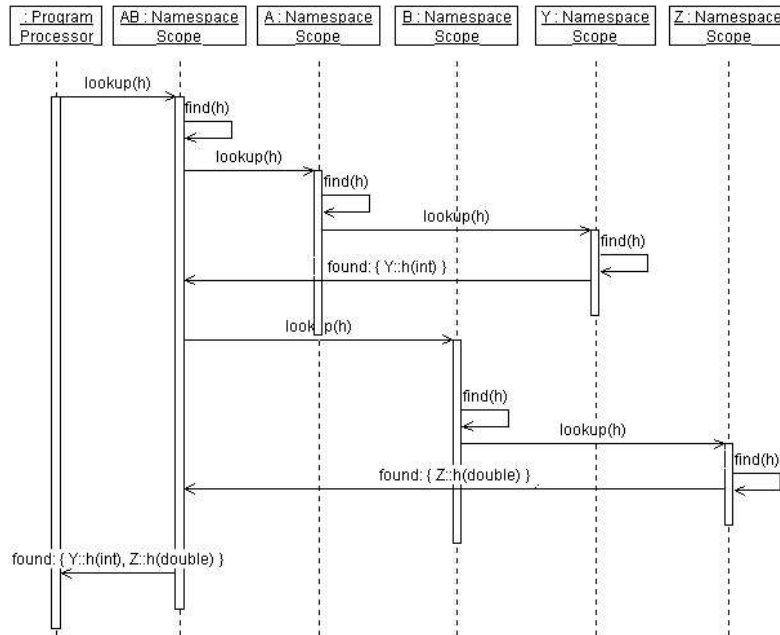
§3.4.3.2/2: Sequence diagram for lookup of `AB::h(16.8)`.

**Paragraph 3**

The examples in this paragraph deal with the situation where the same declaration is found more than once in a namespace search; this occurs when there are two or more routes from the namespace mentioned in the qualifier to the namespace in which the declaration occurs.

Our object diagram here explicitly illustrates this possibility by showing the web of `uses` relationships between the namespaces. Note, however, that since individual declarations are not shown, we cannot tell from the diagram that there is likely to be a clash between the use of `A::a` in namespace `B` and namespace `D`. Perhaps a `uses-a-member-of` relationship might be employed to draw attention to this, if necessary.
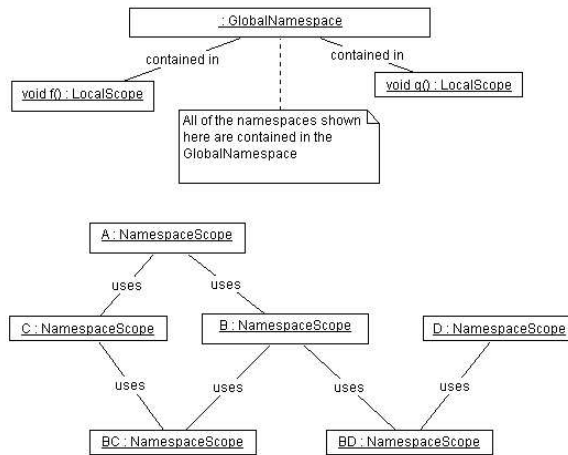


§*3.4.3.2/3: Object diagram* .

Both of the searches are fairly straightforward. In each case the search starts at the namespace used as a qualifier (`BC` and `BD` respectively), and follows back along the path indicated by the `using` directive. We note that in each case the namespace `A` is only searched once, as specified in section 3.4.3.2/2.
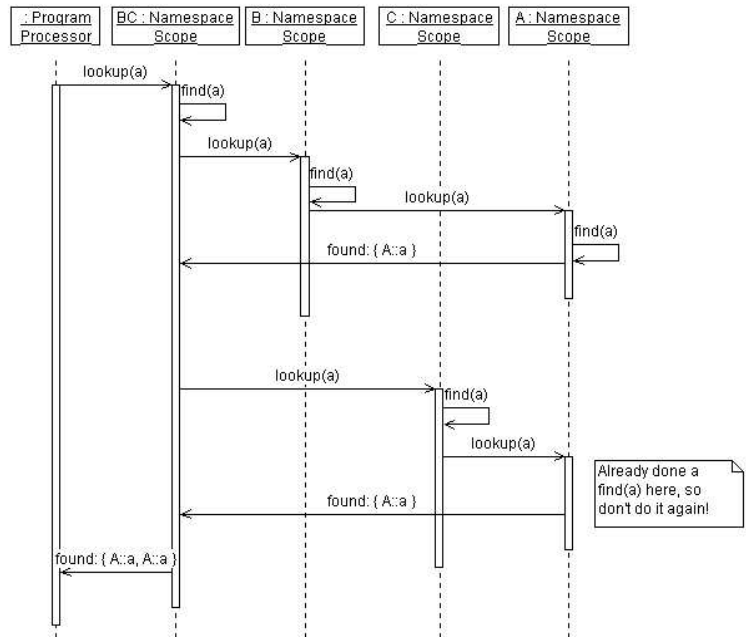
§3.4.3.2/3: Sequence diagram for lookup of BC::a++.
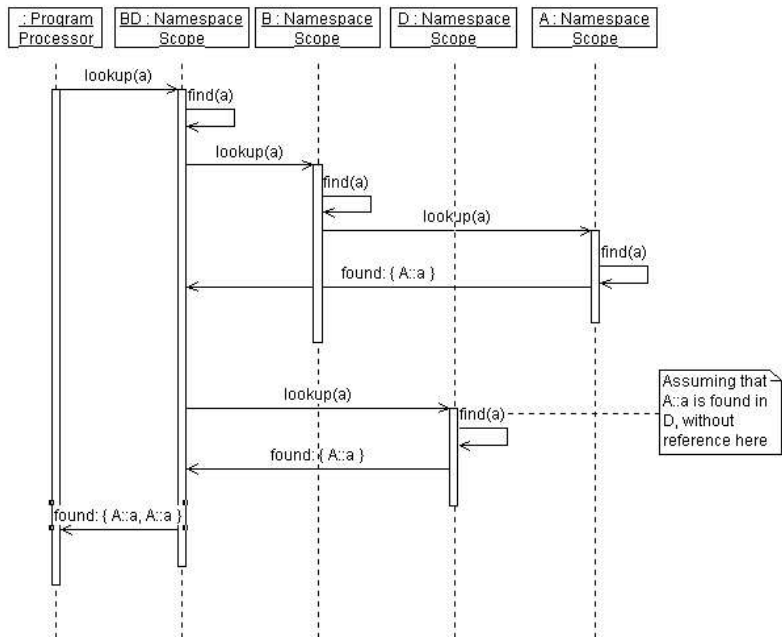


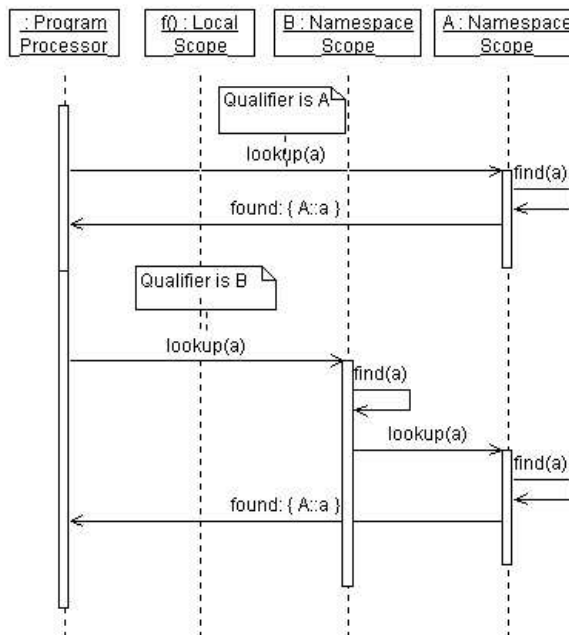§3.4.3.2/3: Sequence diagram for lookup of BD::a++.

**Paragraph 4**

The examples in this paragraph show how being restricted to searching each namespace just once prevents possible ambiguities in those situations where we have a circular uses relationship between namespaces. This circularity is clearly shown in the object diagram.
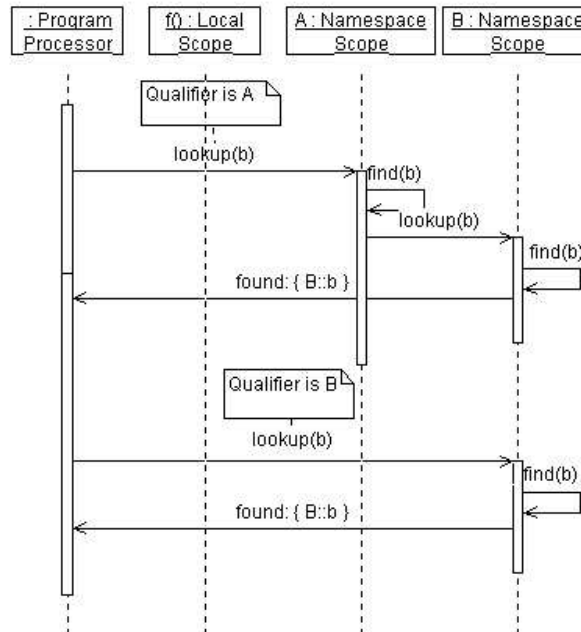


*§3.4.3.2/4: Object diagram .*

The similarities between the searches can easily be seen by comparing the two sequence diagrams here. There is a clear similarity between the search for A::a and B::b, and also between B::a and A::b.



*§3.4.3.2/4: Sequence diagram for lookup of* A::a++ *and* B::a++.

§3.4.3.2/4: Sequence diagram for lookup of A::b++ and B::b++.

**Paragraph 5**

The rules in this paragraph are an elaboration of those in section 3.3.7/2 dealing with re-declarations of the same name, to restrict the validity of such declarations to single namespaces.
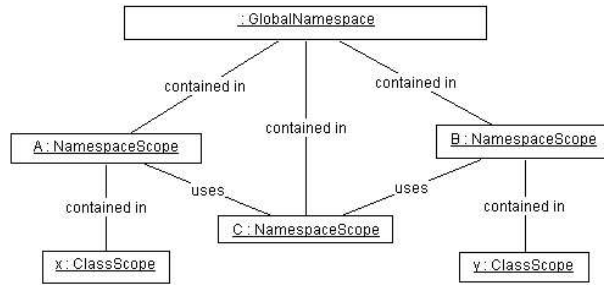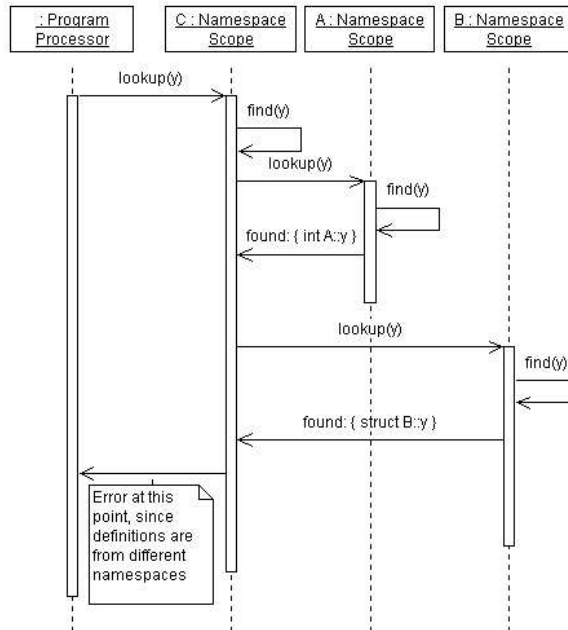


§*3.4.3.2/5: Object diagram .*

Here the lookup proceeds as usual, except now when multiple declarations are found we must be able to distinguish between them.

In the case of the lookup for x it is the job of namespace A to apply the name-hiding rules and return only the non-class definition of x.
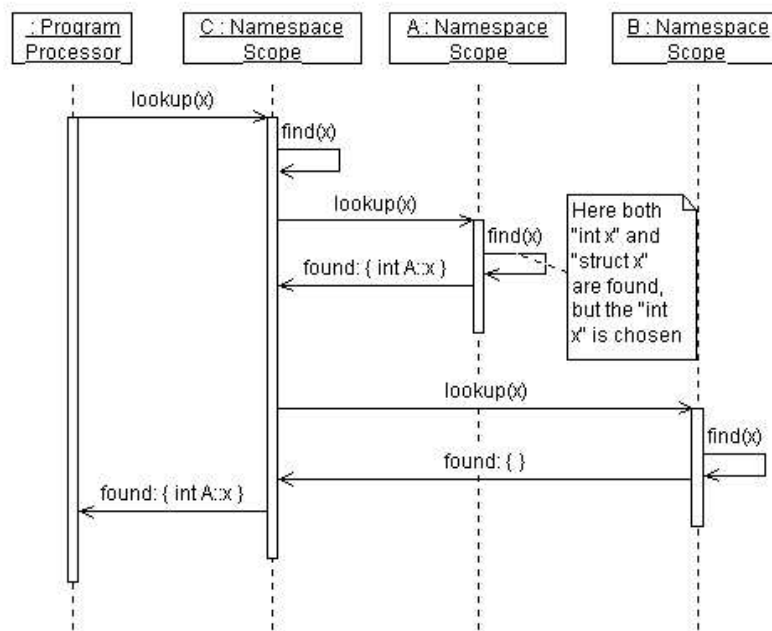


§*3.4.3.2/5: Sequence diagram for lookup of* `C::x`.

The search for y however yields declarations from different namespaces, and it is now the job of the namespace C to realise that these are incompatible. Presumably this forms an adjunct to the collation that namespaces must occasionally perform, as demonstrated in section 3.4.3.2/2. Note, however, that this is restricted to collation and distinguishing incompatible declarations; distinguishing between compatible declarations is the job of overload-resolution, which takes place after name lookup, and is not dealt with here.



§3.4.3.2/5: Sequence diagram for lookup of C::y.

## Paragraph 6

This paragraph contains two separate examples, and hence we have an object diagram and a sequence diagram for each.

In the first case the relationship between the scoping levels is represented by a simple object digram. We represent the logical nesting between the scope levels, including the (correct) nesting of function `f1` in namespace `B`.
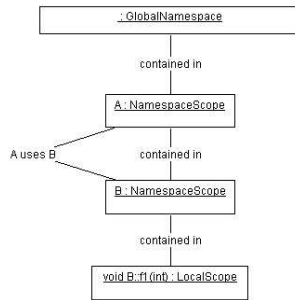


§3.4.3.2/6: Object diagram for the first example.

We have chosen not to represent the incorrect scope corresponding to the definition of `A::f1` in the example, which is presumably contained in namespace `A`. It is not clear at this point whether it would be useful to include a facility in these object diagrams to represent associations resulting from incorrect programs.

The sequence diagram then is the simpler of the two, since we need simply verify that `f1` does not indeed belong to namespace `A`.
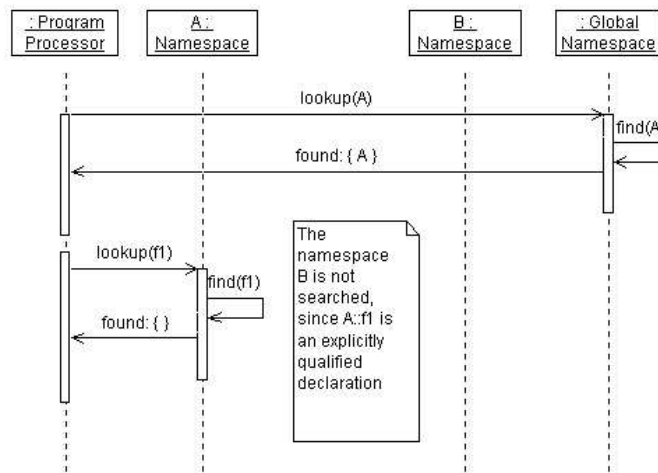


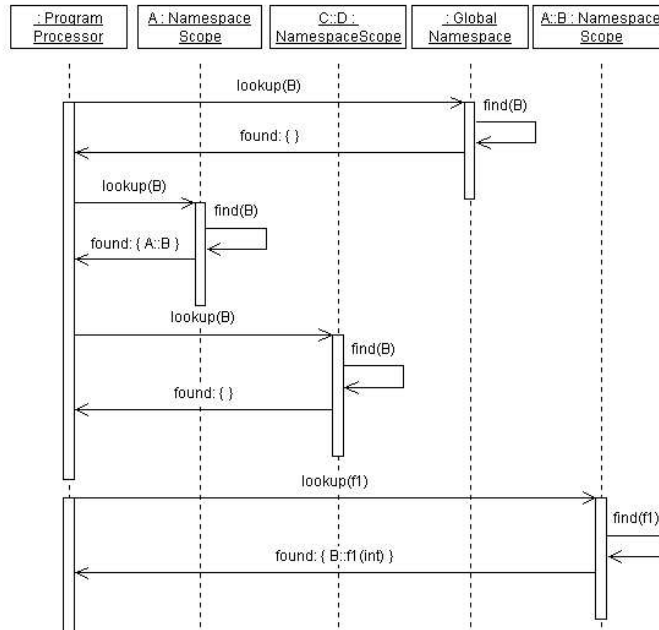§3.4.3.2/6: Sequence diagram for lookup of `A::f1(int)`.

The second example of this paragraph is similar to the first, except now the using directive for namespace A means that the definition of B::f1 is correct.


The object diagram is not particularly helpful to us here, since the potential problem resulting from the physical location of the definition of B::f1 in the GlobalNamespace is not indicated in any way.
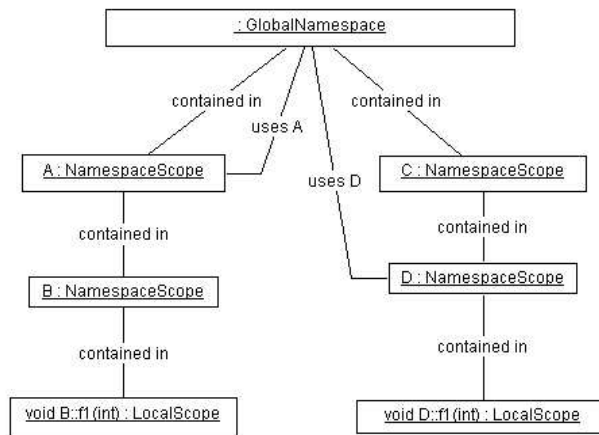


§3.4.3.2/6: Object diagram for the second example.


The sequence diagram does, however, demonstrate exactly why the definition is correct. We can see that the reference to B is resolved in the usual manner, and then subsequent reference to f1 then finds the correct declaration.

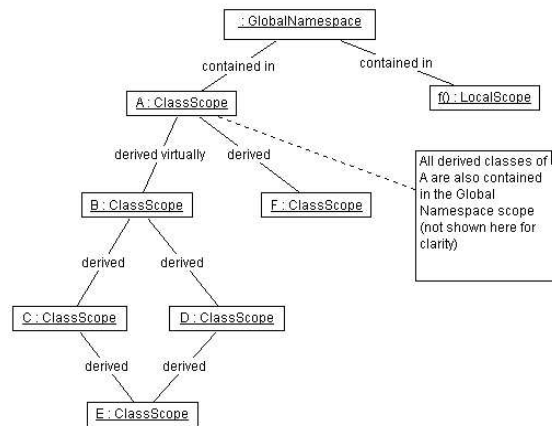§3.4.3.2/6: Sequence diagram for lookup of `B::f1(int)`.

### 3.4.5 Class member access

**Paragraph 4**

The examples in this paragraph deal with the use of an explicit qualifier immediately after a class member access. The example consists of a class hierarchy, with the ultimate base class being the virtually-inherited struct A.
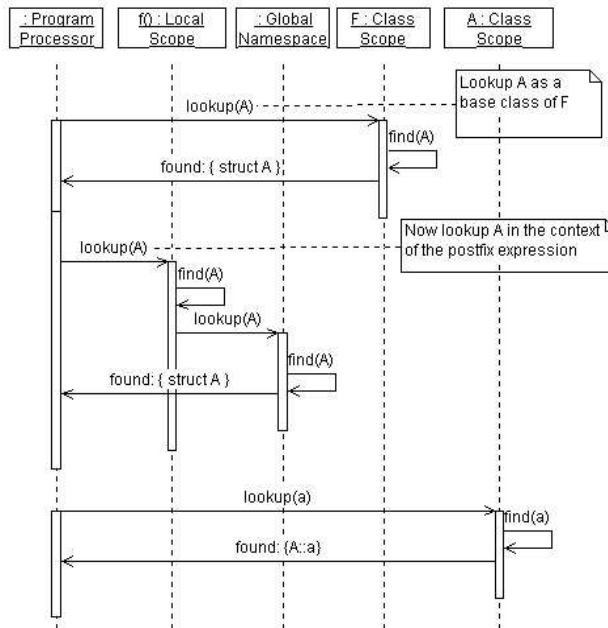


*§3.4.5/4: Object Diagram.*

The sequence diagrams show how the lookup for what is essentially the variable a of struct A proceeds under two different contexts. In each case the class qualifier after the . operator (B and A respectively) is looked up firstly starting in the local scope corresponding to function f(), and then in the scope of the relevant class - E and F respectively.

Note that in the second search in each case we are looking to see if the qualifier represents a viable "view" of the following member. How precisely this is achieved is an implementation matter, and we have chosen in our sequence diagrams to simply show a lookup call being directed at the appropriate derived class. We thus assume that the class can check the validity of the view (either internally or by traversing back up the inheritance hierarchy), and we choose not to represent any further detail in this process.

It is, once again, the responsibility of the program processor to mark the object corresponding to the qualifier (B and A) to indicate that this is a base-class search, and not an ordinary member lookup.
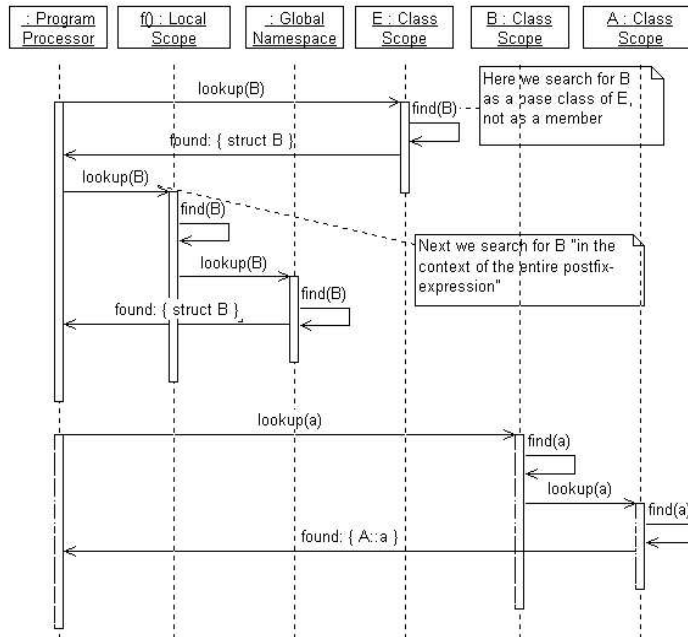
§*3.4.5/4: Sequence diagram for lookup of* `e.B::a`.



§*3.4.5/4: Sequence diagram for lookup of* `f.A::a`.