Object Representation of Scope During Translation

S. C. Dewhurst

AT&T, Summit, New Jersey 07901, USA

ABSTRACT

Languages, such as C++, that combine static type checking with inheritance pose a challenge to the compiler writer: the flexibility of the type and scope system implies that a compiler must be able to cope with data structures of a complexity far exceeding what is needed for traditional languages. Fortunately, they also supply the means for coping with this complexity. This paper describes the strategies devised for representing and manipulating C++ scopes and the storage management techniques employed to make these representations efficient. The techniques have been used to implement a C++ compiler in C++.

1 Introduction

Scope is typically represented at translation time exclusively as an attribute of a name, either explicitly as a field within the name or implicitly by the presence of the name in a data structure that implements the compile-time semantics of scope. The data structures that maintain the relationship between name and scope are traditionally monolithic; that is, all scope information is contained within a single structure. This structure is modified as the program is translated to reflect the changing scope.

This organization is well-suited to the translation of traditional block-structured languages in which the "current" scope changes rather slowly as the program text is analysed. Since scope changes slowly, the modifications of the data structure used to represent it are relatively restrained. However in C++ and a number of other modern languages the combination of static type checking and inheritance produce a scope structure too complex and volatile to be easily and efficiently represented with traditional compiler symbol table structures. The solution developed for the C++ compiler represents scope as an object and the dynamic nature of scope during translation as a dynamic graph of these scope objects. In addition to simplifying the task of maintaining scope information, this approach permits efficient translation-time storage management and provides opportunity to interface to environment tools to save and restore compilation states, and to avoid recompilation of common program pieces.

Section 2 shows how language features affect the structure of scope during translation, section 3 describes the implementation of the object-oriented solution, section 4 shows how this approach allows for efficient translation-time memory management, and section 5 describes some applications this style of memory management makes practicable. Because these techniques were developed for a C++ compiler they are motivated by examples from the C++ language. However, the examples are of minimal complexity and should not require prior knowledge of C++.

2 Effect of Language Features on Scope Structure

For the purposes of the present discussion, a class in C++ can be considered to be simply a record type containing both data and function members. The body of a member function occurs in the scope of the class of which it is a member. Figure 1 has an example of a member function:

```
int i;

class B {
        int i;
        void f();
};

void B::f() { ... = i; }
```

Figure 1

The syntax `B::f` uses the scope operator `::` to indicate that `f` is a member of class `B`. The `i` referenced in the function body of `f` refers to the member `i`, not the global one. Functions can also be defined within a class body:

```
int i;

class B {
        int i;
        void f() { ... = i; }
        friend void f() { ... = i; }
};
```

Figure 2

In Figure 2, class `B` has two members: the integer `i` and the function `f`. The `friend` designation of the non-member function `f` is related to the data hiding features provided by C++ and is beyond the scope of this paper. It is sufficient to note that the friend function is not a member of class `B`, and so the reference to `i` in its body is a reference to the global `i`. Note also that both functions may be referred to through the same identifier because they are defined in different scopes; the member function is in the scope of class `B`, while the non-member function is at global scope. The semantics of the member function `f` in Figure 2 are identical to those of the member function in Figure 1. These examples show that lexical and semantic scope are separate in C++; that is, lexical nesting does not imply semantic nesting.

C++ accomplishes type inheritance through the mechanism of class derivation. The derived class inherits the members of its base class in addition to its own members. However, a member of the base class is hidden in the scope of the derived class if its identifier has been reused in the derived class.

```
class B {
    public:
        int i, j;
};

class D : B {
        int i;
        void f() { i = j; }
};
```

Figure 3

In Figure 3, class `D` is derived from class `B`. The keyword `public`, like `friend`, is related to the data hiding features of the language. The body of member function `f` assigns the value of the base class member `j` to the derived class member `i`. A derived class occurs in the scope of its base class. Note that this "enclosing" scope is unaffected by the scope in which the individual classes are defined:

```
class B { ... };

void g() {
        class D2 : B { ... };
}
```

Figure 4

The enclosing scope of class D2 is still that of B even though D2 is defined in the scope of the function g. These examples show that in C++ there is not necessarily a relationship between enclosing scope and scope of definition of a name, except, of course, that the lifetime of the enclosing scope must at least include that of the scope of definition. This is always the case.

The practical effect of class derivation (type inheritance) on scope is the production of a very complex scope structure and a very volatile notion of the "current" scope. The following example combines the concepts outlined above and will serve to motivate the approaches discussed in the following section:

```
class B {
        int m1();
};

class D1 : B {
        int m2();
};

int f1() {
        class D2 : B {
                friend int f2() { ... }
                int m3() { ... }
        };
}

int D1::m2() { ... }
```

Figure 5

3 Object Representation of Scope

Fortunately, while the features of languages like C++ produce these translation difficulties they also provide the means to solve them through support for the object-oriented programming paradigm. The strategy employed by the C++ compiler is to define a general "scope object" class which contains references to the objects that represent its enclosing scope and routines that implement the semantics of the scope (such as lookup and insertion).

As each scope is entered, a corresponding object is created and its enclosing scope references set to the objects that represent its enclosing scope. When the scope is no longer accessible its scope object is deleted. In this way, as scopes are entered and deleted the compiler maintains a graph of scope objects that reflects the scope structure of the program being translated.† Figure 6 shows the graph that is produced by the example in Figure 5 at the point during which the body of function m3 is being translated.

---

† In the absence of multiple inheritance, each scope object will contain at most one reference to an enclosing scope object and the graph will always be a tree.

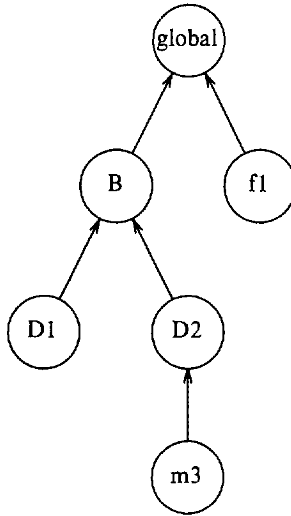

Figure 6

Figure 7 is a finer-detailed view of the structure in Figure 6. It shows the separation of scope of definition and enclosing scope. Names are represented as labeled boxes and scope objects as ellipses. The scope object that is an attribute of a given name is indicated by a solid arrow from the name to the scope object. The dashed arrows show the scope hierarchy illustrated in Figure 6.

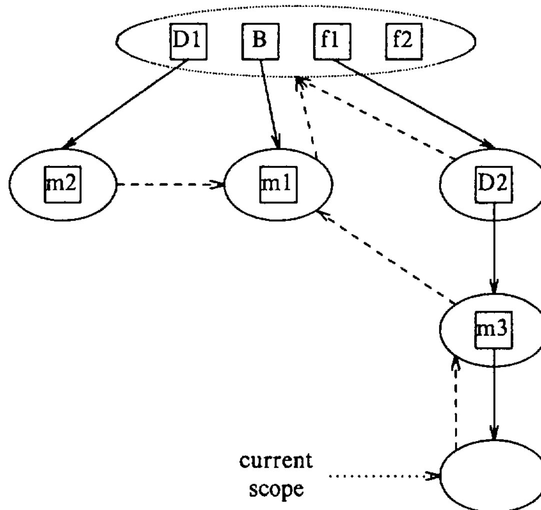

Figure 7

In effect this scheme uses the complexity of the program under translation to build and maintain the scope graph. Thus the burden of complexity is abstracted from the program and the compiler needn't anticipate all the combinations of scoping situations that could arise. For example consider Figure 5 above. Although the scope of definition of `D2` is the scope of `f1` (that is, the name of `D2` is entered in `f1`'s scope) the scope of `f1` is not interposed within the scope

hierarchy formed by D2 and B. Because all scope information about f1 is represented in a single object, the separation of lexical and semantic scope is accomplished simply by positioning f1's scope object in the correct position in the graph. No complex algorithm is needed to bypass the scope of f1 when referencing a name in the scope of D2.

The current scope is represented at any given time by a single node in the scope graph. The current scope is the set of graph nodes reachable from the current node.† For example, the scope current while translating the body of function m3 in Figure 5 is represented in Figure 6 by the list of scope nodes labeled "m3", "D2", "B" and "global". Because the current scope is represented by a single node in the scope graph, the volatile nature of current scope can be handled easily and efficiently by stacking references to nodes. For example, referring again to Figure 5, when the friend function f2 is encountered, the current scope (labeled D2) is pushed on a "lexical scope" stack, and the current scope is set to a new scope node for f2, with global enclosing scope. At the end of translation of f2, the stack is popped to restore the old scope. The definition of the member function m2 at the end of the figure is handled in a similar way; the current (global) scope is pushed and the new current scope is set to a new node for m2, with the parent as D1.

The C++ compiler partitions scopes into three categories: global, class and function. Each category is represented by a class derived from the general scope class, and is tailored to suit the characteristics of the scope in question. The global and class scope types both implement "flat" scopes,† but the global scope type is optimized to handle a larger number of names than the class scope type. The function scope type implements a block-structured scope. Additionally, each type implements a number of scope-specific operations that are not present in the others. The conventional implementation of scope as a monolithic data structure with complex access and insertion rules is replaced by a collection of simpler objects that need concern themselves only with their own structure and the locations of their enclosing scope. This simplicity is a typical advantage of an object-oriented solution.

The decision as to what constitutes a scope is as flexible as the translation task requires. For instance, the C++ compiler does not consider a block within a function as creating a scope; rather, block structure is an attribute of function scope. The rationale for this is that the flexibility of the object approach is not required within a function, where conventional techniques for maintaining block structure are clear and efficient. Additionally, certain translation information, such as activation record layout, is naturally associated with an object that represents an entire function, rather than a collection of block objects.

Another advantage of the object-oriented approach is that of being robust in the face of change. The C++ language is still evolving, and the localization of semantic routines to individual object types allows new semantics to be added without incident. Likely additions to the language, such as multiple inheritance, can be accommodated without significant disruption of the structure of the current compiler.

4 Efficient Translation-Time Memory Management

The logical locality implied by an object also permits an easy implementation of physical locality, either as contiguous memory or in easily accessible locations. That is, if every language feature is associated with a given scope, then the compiler data structure used to translate an instance of that feature can be associated with a given scope object. The lifetime of the data structure should be identical to that of the scope object, just as the lifetime of the feature instance is no greater than that of the scope in which it appears. As an example consider a function definition at global scope. At the exit of the function scope, the scope object of the function may be freed. Note that because the name of the function is not in the function scope, this action will not remove information required for translation to continue. If all the data structures

† In the absence of multiple inheritance the scope is represented by a simple list of nodes.

† That is, a scope in which an identifier always refers to exactly one name.

associated with the function scope object are removed along with it, then the remaining data structures will be left in a consistent state, and translation can continue.

The approach used in the C++ compiler is to associate a different memory space with each active function scope and with the global scope. Class scope objects are allocated in the memory space of the scope that is current at the time of their creation. When a function or global scope object is created, a new memory space is set up and the scope object is allocated within it. All subsequent compiler data structures *within that scope* are allocated in that memory space. At the end of the function or global scope the memory for the scope is freed, removing the scope object from the scope graph and leaving the compiler data structures in a consistent state. Figure 8 shows how the C++ compiler would segment memory for the program example of Figure 5, during the translation of the friend function `f2`. Each dotted boundary encloses a separate memory space associated with an active global or function scope.

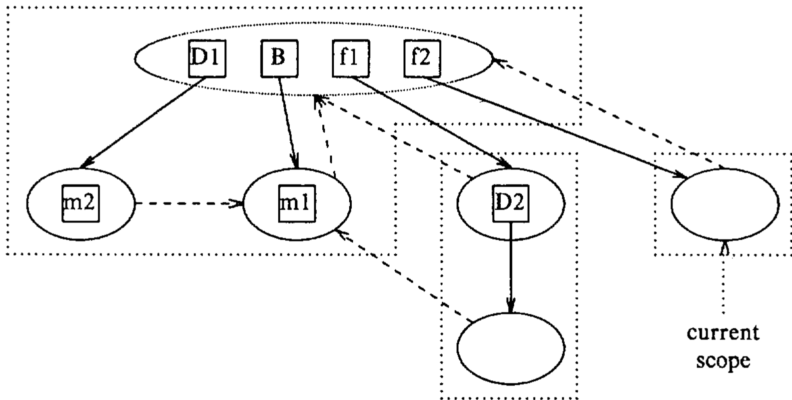

Figure 8

In addition to the efficiency gained by essentially constant time memory management, this segmentation of memory by scope has the advantage of leaving the memory spaces of all active scope objects available at all times. It was shown earlier that a friend function like `f2` of Figure 5, although lexically within function `f1`, is actually defined in the global scope. Consider the difficulties this would present to a stack allocation scheme. The memory associated with global scope would be below that of the function `f1`, and the name of the friend `f2` would be allocated within the memory associated with `f1`. Of course, the name of the friend function would still be entered in the global scope, but on popping the function information off the allocation stack at the end of `f1`, the compiler data structures would be in error, as the friend name will have been removed. More flexible methods of memory management with finer granularity of control can be employed, but these typically have concomitant penalties in performance and complexity.

In associating patterns of memory management with the scope graph, memory management inherits the scope scheme's simplicity while allowing the program under translation to drive the construction of the complex memory management. Each scope object has a memory descriptor. The (unique) allocation routine uses the descriptor of the current scope object to determine the memory space from which to allocate. In the example above, the current descriptor is set to the global descriptor before `f2` is translated and then restored to that of `f1`. In this way, the name of the friend function `f2` is allocated in the global memory space. After the translation of `f2`, the current scope is reset to `f1` and the current memory space is reset from `f1`'s descriptor.

5 Applications of Scope-Segmented Memory Management

A side effect of this memory allocation scheme is the ability to process multiple program files in sequence with little overhead. Many compilers (including early versions of the C++ compiler) process only a single file per invocation because, due to the way memory is managed, cleanup between files would be prohibitively expensive. The C++ compiler solves this problem by creating an object that represents the scope that exists before and between file compilations. Its memory space contains invariant data that is created on compiler initialization (but no names). At the end of each file the global scope object is deleted to re-initialize the compiler. In avoiding loading and initialization for each file, the compiler realizes significant gains in translation time for compilations that comprise several files.

Other, as yet unimplemented, applications which could make use of this strong association between memory space and scope are perhaps best viewed as compiler support for programming environment tools. For example, this organization would simplify an implementation that minimizes recompilation of common file prefixes.

Consider the problem of compiling a sequence of files, each of which has a common initial part. If no external effect is produced during the compilation of the common prefix (such as code generation) then the effect of the translation to that point is to move the compiler from its initial configuration to a given compilation state. If this state can be restored easily, the common part need be compiled only once for the sequence of files.

In most compilers the state of compilation is represented entirely within the compiler's data structures. If the memory allocation scheme described above is used, these are contained within the memory spaces of a collection of scope objects. Thus, just as earlier we were able to restore the initial compiler state by deleting all scope objects but the initial "between file" object, it is possible to restore any intermediate compilation state through judicious segmentation and deletion of scope objects. (Note that there needn't be a one-to-one relationship between scopes and scope objects.) The ease with which this can be accomplished is highly language dependent. For example, the C++ compiler could easily implement delayed code generation (expanding the class of code prefixes it could handle) because much of that capability is required for inline function expansion. On the other hand, in C++ the definition of a name may be begun at one point in the compilation and added to at a remote point. (For example, a function prototype may add default argument initializers.) In order to handle prefixes containing definitions of this kind it would be necessary to record and undo effects of this sort that occur after the prefix.

Once the case of the single common prefix can be handled, a number of additional opportunities arise. Perhaps the most obvious of these is the stacking of compilation states. In this way the prefix dependancies of a set of files to be compiled can be arranged into a DAG and the files compiled in such an order that by pushing and popping the scope objects representing compilation states minimal recompilation is performed. Additional gains can be obtained by the ability to save an external representation of a compilation state in a program database.

6 Summary

The C++ compiler deals with the complex and volatile scope structures that arise during translation by abstraction to a graph structure; a scope is represented as a node and an enclosing scope relationship as an edge. The logical encapsulation of scope as object eases the task of representing the complex nature of scope in languages like C++, and permits both reduction in complexity and increased flexibility. In extending the logical encapsulation implied by an object to physical encapsulation, the C++ compiler implements efficient and simple translation-time memory management. In effect, the compiler uses the complexity of the program being translated to guide the building and maintenance of a complex data structure composed of simple components.

The simplicity of the resultant compilation structures permits further optimization of the translation process, including multiple file compilation and intelligent interface to program environment tools in order to avoid recompilation of common code pieces.

7 Acknowledgements

The author wishes to thank Bjarne Stroustrup for invaluable advice on organization and content of this paper, and Kathy Stark, Laura Eaves and Barbara Moo for many valuable discussions and comments.

[1] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.