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:
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:

```cpp
int i;

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

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

Figure 1

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.

```cpp
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: