Modeling the C++ Object Model

An Application of an Abstract Object Model

Alan Snyder
Hewlett-Packard Laboratories
P.O. Box 10490, Palo Alto CA 94303-0969

Abstract

We are developing an abstract model to provide a framework for comparing the object models of various systems, ranging from object-oriented programming languages to distributed object architectures. Our purpose is to facilitate communication among researchers and developers, improve the general understanding of object systems, and suggest opportunities for technological convergence. This paper describes the application of the abstract object model to the C++ programming language. We give an overview of the abstract object model and illustrate its use in modeling C++ objects. Several modeling alternatives are discussed and evaluated, which reveal anomalies in the C++ language. We conclude by characterizing those aspects of the C++ object model that distinguish C++ from other object systems.

Introduction

Concepts originating in object-oriented programming languages are appearing in many variations in different technologies, ranging from distributed systems to user interfaces. In previous work [9], we identified what we believe are the essential concepts underlying these variations. Our current effort involves the creation of an abstract object model, which provides more precise definitions of the essential concepts. The purpose of the model is to serve as a framework for comparing the object models of different technologies to identify common properties, highlight differences, and suggest opportunities for technological convergence. The object model is being developed by applying it in turn to five major technologies of interest to Hewlett-Packard. This paper reports on the first such application, to the C++ programming language [5].

In this paper, we present an overview of the abstract object model and illustrate its use in modeling C++ objects. The presentation concentrates on aspects of the model where the application to C++ raises issues either about the model or about C++ itself. We begin by reviewing the relevant aspects of C++.

A Brief Overview of C++ Objects

In this section, we review the object-related aspects of C++. (Readers familiar with C++ should skim this section for terminology used in later sections.) For brevity, we omit certain features of C++ that do not significantly affect the presented material: access control, virtual base classes, and references.[1] In this section, all terms are C++ terms, not object model terms. For example, the term *object* means C++ object, which is a region of storage. Note that our description of C++ is abstract; it is not an implementation model.

1. For the purposes of this discussion, references are equivalent to pointers. They are called pointers herein.

The principal C++ construct related to object-oriented programming is the *class*. A class serves two roles, as a lexical scope and as a type. As a lexical scope, a class defines a set of immutable bindings (called *members*) between names and certain kinds of entities, which include data declarations and various kinds of literals (types, enumerations, and functions). The members of a class have distinct names, except for function members, which must be distinguishable by their names and their declared argument types (functions distinguished only by their declared argument types are called *overloaded functions*). A class provides lexical context for the definitions (bodies) of its function members and for nested class and function definitions; the enclosed definitions can reference the members of the class by name. Function members can be of several varieties: ordinary, static, virtual, and pure virtual. Data declarations can be of two varieties: ordinary and static.

A class also defines a type, which is a pattern for instantiating objects, called *class instances*. A class instance is a compound object: it consists of multiple subobjects, called *components*. (The C++ literature uses the term *member* for these subobjects; we introduce the term *component* to avoid confusion with class members.) The components of a class instance are determined by the class. For a simple class (not defined using derivation), there is one instance component for each data declaration, each ordinary function member, and each virtual function member. Function components can be viewed as closures unique to the instance: they have direct access to the components of that instance, and can refer to the instance itself using the variable *this*.

Class members are named directly (in their scope) or using the notation '*class*::*name*'. Instance components are accessed using the notation '*instance.member*', where *instance* is an expression denoting a class instance and *member* names a class member. A complex example is $a.A::B::x$, where x is a member of the class that is the B member of class A, and a is an instance of a class (such as $A::B$) containing a corresponding x component. Within a function component, components of *this* are accessed using the member name alone. There are many restrictions on the use of class members and instance components, but they are not important for this presentation.

For the purposes of the object model, we treat a pure virtual function as a virtual function with a distinguished definition (that cannot be referenced). We ignore the remaining kinds of class members (types, enumerations, static functions, and static data declarations) henceforth, as they do not impact the object model.

Figure 1 shows a class named A that defines two members: a data declaration (for an integer object) named x and an ordinary function named f. The second line provides the full definition of f. The third line creates an instance of A called a. It has two components, an integer object named x and a function named f. The following lines access the components of a. Note that the component function f of a accesses both the x and f components of a by name. The diagrams illustrate the class A and the instance a. We use shaded boxes to denote class members and unshaded boxes to denote instance components. We introduce the notation A/f to refer to the function member f of the class A and the cor-

responding function components. (Our examples will not involve overloaded functions.)

A class can be defined by deriving from one or more *base classes* (we use the term base class to mean *direct* base class, unless explicitly indicated otherwise). The effect of derivation on the derived class lexical scope is similar to nested scopes: the derived class lexical scope includes not only the members it defines directly, but also any member of a base class that is neither redefined in the derived class nor ambiguous in multiple base classes. The effect of derivation on instances of a derived class is composition: an instance of a derived class contains not only the components corresponding to the members defined directly in the derived class, but also one unnamed instance component of each base class. We call these unnamed components *base components*.

Figure 2 illustrates class derivation. The class D is derived from the class B. Class B's lexical scope has three members: x and y (both data declarations for integer objects), and f (an ordinary function). Class D defines two members: x (a data declaration) and f (an ordinary function). Class D's lexical scope includes x and f, but also includes y (which it inherits from B). An instance of class D contains three components: x (an integer object), f (a function component), and an unnamed instance of class B. The class B base component contains three components: x and y (integer objects) and f (a function component). The diagram illustrates the structure of an instance of class D.

```
class A { public: int x; void f (int);};
void A::f (int g) {x = g; f (g+1);};
A a;
a.x = 3;
a.f (4);
```

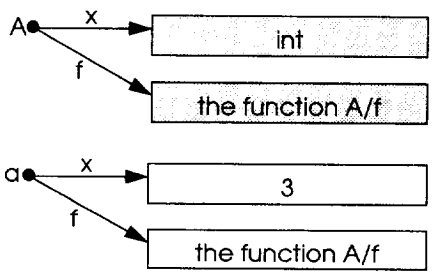

Figure 1. A class and a class instance.

```
class B { public: int x, y; void f (int);};
void B::f (int g) {x = g; f (g+1);};
class D : public B { public: int x; void f (int);};
void D::f (int g) {x = g; f (g+2);};
D d;
D* pd = &d; // create a pointer to d
B* pb = pd; // type conversion
```

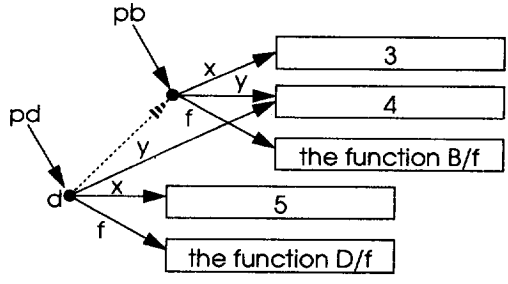

Figure 2. A class defined using derivation.

A function component can access only the components of the instance of its defining class. For example, the function *B/f* in Figure 2 can access only the components of the *B* base component; its variable *this* is equivalent to *pb*. In contrast, in the function *D/f*, *this* is equivalent to *pd*. Although redefined base class members are not part of the derived class lexical scope, they can be accessed from the derived class scope (and elsewhere) using explicit qualification. For example, the function *D/f* in Figure 2 can access the *x* component of the *B* base component using the name *B::x*.

An implicit type conversion is defined from type 'pointer to derived class' to type 'pointer to base class', for each base class. Its effect is to convert a pointer to the derived class instance to a pointer to the corresponding base component. Figure 2 illustrates this conversion: the pointer *pd* to the *D* instance is converted to the pointer *pb* to the *B* base component. This conversion achieves the *effect* of inclusion polymorphism [3]: a pointer to a *D* instance can be passed as an argument to a function expecting a pointer to a *B* instance.

A function member declared *virtual* produces a different instance structure. A virtual function component *overrides* (takes precedence over) any direct or indirect base class function components of the same name and type. (The actual rules are more complex.) Figure 3 shows the effect of declaring *f* to be virtual: in the lexical scope accessed from *pb*, *f* now denotes the component function *D/f* (instead of *B/f*). Virtual functions allow specialization to be effective with inclusion polymorphism: the derived class function component will be invoked even from a context where the object is known as an instance of the base class. For example, the function *test* in Figure 3 invokes *D/f* when its argument is *pb* (or *pd*), even though the variable *p* is of type 'pointer to *B*'.

Explicit qualification has the effect of suppressing virtuality: the expression *d.B::f* refers to the *B/f* component of *d*, not the *D/f* component. Thus, the :: operation is *not* simply a scoping operator.

```
class B { public: int x, y; virtual void f (int);};
void B::f (int g) {x = g; f (g+1);};
class D : public B { public: int x; void f (int);};
void D::f (int g) {x = g; f (g+2);};
D d;
D* pd = &d; // create a pointer to d
B* pb = pd; // type conversion
void test (B* p) {p→f(0);}
```

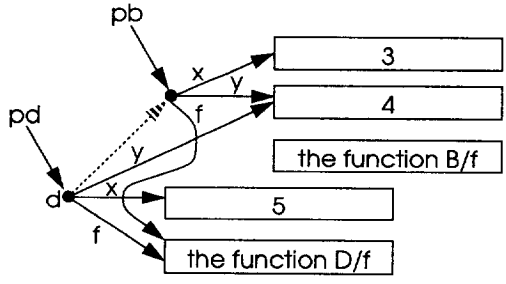

Figure 3. Virtual functions.

The Abstract Object Model

In this section, we present an overview of the abstract object model. (A more complete presentation of an earlier version can be found in [8].) We concentrate on those aspects of the model that raise the most interesting issues when modeling C++. Terms, such as *object*, used in this section refer to model concepts, as distinct from C++ concepts.

The model is based on the following concepts we deem the essential concepts of objects (further information on these concepts can be found in [9]):

- An object explicitly embodies an abstraction that is characterized by services.
- Clients request services; a request identifies an operation and zero or more objects.
- Operations can be generic: an operation can be uniformly performed on a range of objects with visibly different behavior.
- Objects are distinguishable, and new objects can be created.
- Objects can be classified by their services, forming an interface hierarchy.
- Objects can share implementation, either in full (class instances) or in part (implementation inheritance).

The abstract object model is a partial description of the behavior of a computational system. To model a particular system (such as C++), one defines a *concrete object model* for that system. A concrete object model may differ from the abstract object model in several ways. It may *elaborate* the abstract object model by making it more specific, for example, by defining the form of request parameters or the language used to specify types. It may *populate* the model by introducing specific instances of entities defined by the model, for example, specific objects, specific operations, or specific types. It may also *restrict* the model by eliminating entities or placing additional restrictions on their use.

The abstract object model postulates a set of *clients* that issue requests for service and a set of *objects* that perform services. (An object can be a client, but clients need not be objects.) A request is an *event*: a unique occurrence during the execution of the computational system. A request has associated information, which consists of an *operation* and zero or more parameter *values*. A value may identify an object; such a value is called an *object name*. The operation identifies the service to be performed; the parameters provide the additional information needed to specify the intended behavior. A client issues a request by evaluating a *request form*; each evaluation of a request form results in a new *request*. The mapping of a request form to a request is called *spelling*.

The abstract object model takes the perspective of the *generalized* object models found in the Common Lisp Object System [1] and the Iris database [6]. In a *classical* object model (as in Smalltalk), each request contains a distinguished parameter that identifies the *target* object of the request, which then controls the interpretation of the request (called a *message*). A generalized object model allows multiple parameters denoting objects to influence the interpretation of the request. The generalized model includes classical models as a special case.

An operation is simply an identifiable entity. Its purpose is to characterize sets of requests with similar intended semantics. To allow operations to have associated semantics in a computational system, we believe it is necessary to allow developers to *create* operations (i.e., uniquely allocate an operation for a particular use). An operation is named in a request form using an *operation name*.

A *generic operation* is one that can be uniformly requested on objects with different implementations, producing observably different behavior. Intuitively, a generic operation is implemented by multiple programs, from which a single program is selected dynamically for each request.

A request is performed by transforming it into a *method invocation*. This transformation is called *binding*. A method invocation identifies a *method*, a collection of *method parameters*, and an *execution engine*. A method is a program. The method parameters are values, possibly from a different space than request parameters. The execution engine interprets the method in a dynamic context containing the method parameters; upon completion, a *result* is returned to the client. Execution of a method may alter the state of the computational system. The input to binding consists of a request (an operation and parameter values) and a request context. The request context supports the option of client-specific behavior.

To review, a client issues a request by evaluating a *request form*, which contains an *operation name*. The request form is mapped to a *request* by a process called *spelling*. The request identifies the *operation* and some parameter *values*, which may be *object names*. The request is mapped to a *method invocation* by a process called *binding*. This two-stage processing model is shown in Figure 4.

The intermediate stage of requests serves to capture the essential information provided by a client when requesting a service. Spelling and binding serve distinct purposes. Spelling is a convenience for clients: it provides flexibility in how things are written. Binding captures a fundamental property of object system: the provision of multiple implementations for a single semantic abstraction.

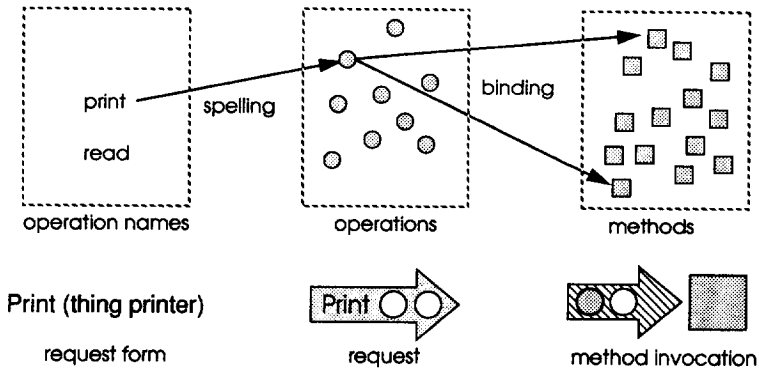

Figure 4. Spelling and binding.

The model defines the concept of a *meaningful request* to capture the notion that not all possible requests are sensible. (Many systems formalize this notion in a type system and verify type correctness using static or dynamic type checking.) *Meaningful* is a boolean predicate on requests that is defined in each model of a particular object system.

For convenience, several subsidiary concepts are defined: An *operation signature* is a description of the parameter values that are meaningful in requests that identify a particular operation. Effectively, operation signatures are a factoring of the *meaningful* predicate by operation. An operation signature may also constrain the results of the corresponding requests. A *type* is a boolean predicate on values that can be used in operation signatures. A relation called *subtype* is defined over types: a type a is a subtype of a type b if any value that satisfies type a necessarily satisfies type b. An *interface* is a (partial) description of an object that describes circumstances under which it is meaningful for an object to be named as a request parameter; in effect, an interface describes valid uses of an object, from a client perspective. An interface can be a type, called an *interface type*.

Examples: The signature of the *push* operation might specify that the first parameter must be a stack (an object name that satisfies a type *stack*) and the second parameter must be an integer (a value that satisfies a type *integer*). The stack interface (which describes stack objects) might specify that a stack can appear as the first parameter to the *push* and *pop* operations.

Using the Abstract Object Model

Using the abstract model to describe an existing system is a subjective process. The modeler makes choices that can be evaluated only using subjective criteria, such as simplicity, comprehensiveness, and utility. For example, consider a model in which there is exactly one operation, identified in every request. In this model, the operation conveys no information; the identification of the requested service would have to be communicated either as a parameter value or via the request context. We consider this model poor because it fails to use operations effectively to characterize sets of requests.

One aspect of our work has been to accumulate a set of more specific evaluation criteria. The criteria were developed as a way of justifying and explaining choices that were made intuitively. They are guidelines, not absolute requirements.

The criteria are:

- A typical request form should have the property that all evaluations of the request form issue requests that identify the same operation. In other words, most request forms statically identify an operation. This criterion reflects the intuition that each request form has an associated semantics that corresponds to the expectations of the client.[2]

2. We expect the primary exception to this criterion to be request forms that involve an *operation variable*. An operation variable is a distinct subform whose evaluation results in the identification of an operation and whose evaluation is independent of other parts of the request form.

- There should be generic operations (operations with multiple methods invocable from a single request form). Our intent is to exclude a model in which operations are identified with methods: such a model would incorporate binding into evaluation.
- Operation signatures should be useful (i.e., not too permissive); they should reflect inherent system structure. (The model with a single operation fails this criterion because the signature of the universal operation permits *any* collection of parameters.)
- Operations should distinguish request forms whose potential behaviors are disjoint. For example, two methods that can never be invoked from the same request form should be implementations of distinct operations.
- Operations should characterize objects, in terms of their legitimate use in requests. For example, in most object systems, an object supports a specific set of operations, meaning that it can appear as the target parameter in requests that identify those operations.
- The use of the request context in binding should be minimized.

Modeling C++ Objects

In this section, we present a model of C++ objects using the abstract object model. We concentrate on four aspects of modeling: identifying the objects, operations, values, and types. We discuss several issues that arose in developing the model, and their implications both on the abstract object model and on C++ itself. In this section we must refer to both abstract object model concepts and C++ concepts, some of which have the same terms. Where context is inadequate to avoid confusion, we will use terms like "C++ object" to refer to the C++ concept and "object" to refer to the abstract object model concept.

A fundamental modeling issue is tension between accuracy and expressiveness. Although C++ supports object-oriented programming, it is not a pure object-oriented language. It is not surprising, therefore, that we sometimes had to choose between a model that describes the full semantics of C++ and one that better captures the 'spirit' of objects, but fails to handle certain corner cases of the language. We have taken the latter option. We argue that this approach is better for the purpose of characterizing the C++ object model and comparing it to the object models of other systems. We would not take the same approach if our goal were to create a formal definition of C++.

What are the Objects?

The first issue: what are the objects? There are two independent choices. The first choice is whether all C++ objects are objects, or just class instances. We have chosen to model only class instances as objects, because (as we will describe) only class instances support generic operations.

The second choice is whether an instance of a derived class is modeled as a single object, or whether each base component is modeled as a separate object. This

choice is more significant, and deserves a fuller explanation. (In both cases, a *named* instance component *is* modeled as a distinct object.)

The multi-object model (shown in Figure 5a) models each base component as a distinct object. The implication is that a pointer to a base component like *pb* is a *different value* than a pointer to the derived class instance like *pd*. These values have different types. Because they are different *values*, they can affect binding. For example, if class *B* and class *D* both define an ordinary function member *f*, then the fact that a request (*f pb*) invokes *B*::*f* and a request (*f pd*) invokes *D*::*f* can be understood as a consequence of the two requests identifying the same operation, but different parameter values.

The monolithic object model (shown in Figure 5b) models base components as lacking a separate identity. The implication is that a pointer to a base component like *pb* is modeled as *the same value* as a pointer to the derived class instance like *pd*. The fact that these pointers have different semantics cannot be explained based on values, but must be explained based on static types of expressions. Specifically, we use the static type of the target object expression to map the operation name *f* into *distinct operations*, which we will label as *B*::*f* and *D*::*f*. A request (*B*::*f pd*) invokes *B*::*f* and a request (*D*::*f pd*) invokes *D*::*f*.

The monolithic object model forces us to model *B*::*f* and *D*::*f* as distinct operations. We argue that this modeling is appropriate, because in C++ no single request form can invoke both of these functions (ignoring pointers to class members, which are *operation variables*). We prefer the monolithic object model, both because it is simpler, and because it is more consistent with the "mainstream" concept of object. (In most object systems, instances of classes defined using inheritance do not reveal themselves as consisting of distinguishable parts.) Furthermore, modeling *pb* and *pd* as the same value is consistent with C++ pointer comparison, which reports these pointers as being equal after an implicit type conversion.

The disadvantage of the monolithic object model is that it fails to handle a corner case of the language where an object visibly contains more than one base component of the same type. Clearly, multiple base components of the same type cannot be distinguished by type, but only by value. This situation involves a distinctive use of C++ multiple inheritance, illustrated in Figure 6. Class *D* is derived from classes *B* and *C*, each of which are derived from class *A*. A *D* in-

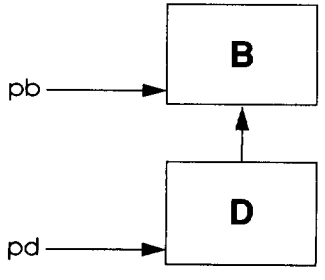

Figure 5a. Multi-object model.

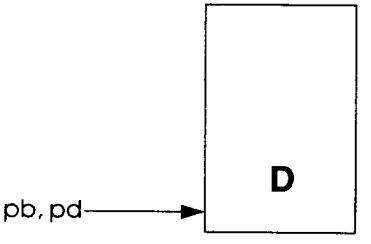

Figure 5b. Monolithic object model.

stance contains two base components of class *A*. Using two levels of type conversion (one of which must be explicit), a client holding a pointer to the *D* instance (*pd*) can obtain pointers to either *A* instance (*pa1* or *pa2*).

To reflect this case accurately, we must either model the *A* components as separate objects, or we must extend the abstract object model to include a concept of object port (where a single object can have multiple ports). Our position is that the extra complexity needed to handle this case is not justified. This position rests upon an assumption that the above situation is unusual, which we defend based on the fact that the client must use explicit type conversions to obtain access to the component instances.

To summarize, we model only C++ class instances as objects, and we model instances of derived classes as monolithic (base components are *not* separate objects).

What are the Operations?

The second modeling issue is to determine the space of operations. One possibility is to model only virtual function members as operations, because (as we will show) they correspond to generic operations. Alternatively, one could model all nonstatic function members as operations, or all function members. Our choice is to model *all* C++ functions as operations. This choice is consistent with the generalized object model approach. The abstract object model does not require that *all* operations be generic. Having made this choice, we model *all* C++ function invocations as request forms (one exception is introduced below); the various cases are shown in Figure 7.

How do C++ functions map to operations? The primary modeling issue is to define operations that are (potentially) generic. Recall the key characteristics of a generic operation: (1) it can have multiple methods selected based on request parameters, and (2) the methods of a generic operation can be invoked from a single request form. To model generic operations, multiple functions should map to the same operation: they are modeled as different methods for that operation. C++ provides two candidates for generic operations: overloaded functions and virtual function members.[3]

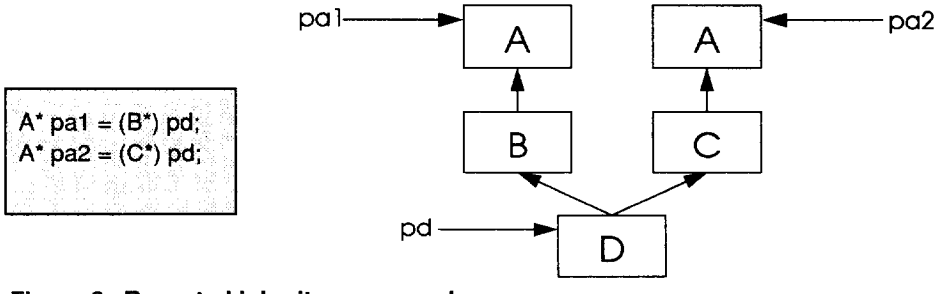

Figure 6. Repeated inheritance example.

3. Other kinds of functions, such as non-virtual member functions, cannot be generic operations because they are statically identified in all request forms (ignoring pointers to functions).

C++ overloaded functions are functions that have the same name in the same lexical scope, but are distinguished by their formal argument types. For each invocation that names a set of overloaded functions, a specific function is selected from that set at compile-time based on the static types of the argument expressions. Therefore, a single request form *always* invokes the same function, which contradicts the key characteristics of generic operations. Thus, we do *not* model overloaded functions as generic operations. Specifically, we model each function in a set of overloaded functions as a *distinct* operation. We model overloaded function resolution as part of *spelling*: it is a convenience for programmers, not support for object-oriented programming.[4]

C++ virtual function members are appropriately modeled as generic operations because they satisfy the key characteristics of generic operations: a single request form naming a virtual function member can invoke function components that correspond to *different* function definitions. For example, as shown in Figure 3, an invocation of *f* using a pointer of type 'pointer to *B*' might invoke either *B/f* or *D/f*, depending upon whether the pointer points to an instance of *B* or an instance of *D*, respectively.

We model a generic operation as a single operation with multiple methods. In the simple case exemplified by Figure 3, the two functions *B/f* and *D/f* both correspond to a single operation ϕ; they are two methods for the same operation. A request form naming either function member issues a request for operation ϕ. The request (ϕ α) is bound to the appropriate function based on the parameter value α, which is an object name that identifies an instance of *B* or a class directly or indirectly derived from *B*.

A more challenging example is shown in Figure 8a. Class *E* is a derived class with two base classes *B* and *D*, which are derived from classes *A* and *C*, respectively. Each of the five classes defines a virtual function member *f* with no arguments. There are five functions named *f*; how many operations are there?

ordinary functions	f (e1, e2, ...)
pointers to functions	e (e1, e2, ...)
static function members	C::f (e1, e2, ...)
nonstatic function members	o.f (e1, e2, ...)
pointers to class function members	o.e (e1, e2, ...)

f ∈ identifier
C ∈ class name
e ∈ expression – identifier
e1, e2 ∈ expression
o ∈ expression (denoting a class instance)

Figure 7. C++ request forms.

4. As an aside, if we wanted to model overloaded functions as generic operations, we would be forced to abandon the monolithic object model. Overloaded function resolution is based on static expression types. To model overloaded function resolution as part of *binding*, the parameter values must capture the static type information. The monolithic object model of values discards the static type information associated with pointers to base components.

Based on the previous example, it is clear that *A/f* and *B/f* are one operation (1) and that *C/f* and *D/f* are one operation (2). Furthermore, we argue that operations 1 and 2 are distinct: no single request form can invoke both *A/f* and *C/f* (ignoring pointers to class members). However, the logic of the previous example also argues that *E/f* is the same operation as both *A/f* and *C/f*: an invocation using a 'pointer to *A*' can invoke *E/f* (if the pointer denotes an *E* instance); an invocation using a 'pointer to *C*' also can invoke *E/f*.

Our solution to this problem is to model *E/f* as a method for *two* operations. (One can imagine that *E* actually defines two equivalent functions.) A potential problem with this solution is that an invocation of *f* using a 'pointer to *E*' is ambiguous: which operation does the request identify? Fortunately, the ambiguity is unimportant: either operation could be identified; the rules of C++ exclude as ambiguous any case where the choice would make a difference in the code that is executed (e.g., the following example).

A related example (shown in Figure 8b) illustrates an anomaly in C++ that was exposed during the creation of the model. We have changed the example by removing the definitions of *f* members in classes *B* and *E*. The name *f* is now ambiguous in class *E*, because it might refer to either *A/f* or *D/f*; an invocation of *f* using a 'pointer to *E*' is illegal in C++. The anomaly is that using the class scoping operator to disambiguate the two operations has undesirable effects, because explicit qualification suppresses virtuality. Although an invocation of *B::f* using a 'pointer to *E*' *will* invoke *A/f*, the client code is not resilient: if a definition of *f* is added to *E*, the invocation will still invoke *A/f*, not *E/f*. A better solution is to disambiguate using type conversion: converting a 'pointer to *E*' to a 'pointer to *B*' yields a value upon which the name *f* is unambiguous. This anomaly is not a problem for the object model; it is a shortcoming of C++ that in the lexical scope of class *E* there is no way to *spell* either operation.[5]

Modeling virtual function members as generic operations again fails to model the full semantics of C++. Specifically, it fails to model the ability of clients to use explicit qualification to name individual functions (such as *A/f*, *C/f*, and *D/f* in Figure 8b). For example, an invocation that names *C::f* cannot be modeled as a request form in this model: the operation corresponding to *C::f* is the same op-

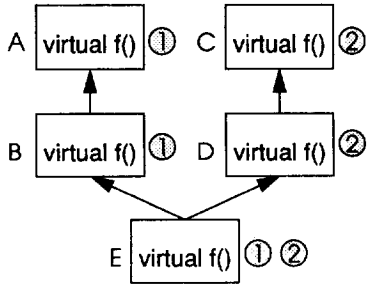

Figure 8a. Virtual function members.

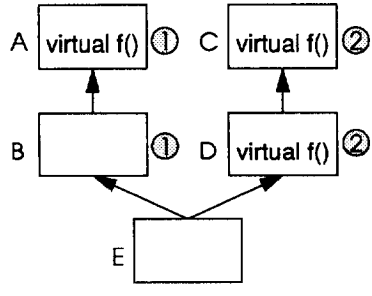

Figure 8b. An anomalous example.

5. Proposals to correct this shortcoming were rejected by the ANSI C++ committee because a programmer can work around any problems by defining additional classes.

eration corresponding to *D/f*; the operation does not provide enough information to allow the client to invoke *either* function on the *E* instance (in distinct invocations). Although we could use the request context to provide the necessary information, we prefer to model such invocations as *direct method invocations*, rather than as request forms. We believe this choice is consistent with the spirit of the language, based on the recommendation of the language designer that such invocations be used *only* within methods, and not in client code [5, p. 210].

The modeling of C++ functions as operations is summarized as follows: Each C++ function (executable or pure) is a distinct operation, except for a virtual function member that overrides one or more base class virtual function members, which instead provides an additional method for each of the original operations. An operation in C++ is identified by a triple: a lexical scope (such as a file or a class definition), a function name, and the formal argument types (suitably canonicalized to reflect C++ overloaded function resolution). For an operation corresponding to a virtual function member, the identifying lexical scope is the "most base" class defining the virtual function member, i.e., the root of the class derivation tree where the virtual function member is introduced.

Operations are *values* in C++: they may be used as request parameters. Operations that correspond to ordinary functions and static function members are values whose types have the form 'pointer to function ...' (the elision describes the argument and result types); these operations have exactly one method each. Operations corresponding to nonstatic function members are values of type 'pointer to class ... function member ...' (the first elision names a class, the second describes argument and result types). Pointers to class function members identify a class member, *not* an instance component; an instance must be supplied when the function is invoked. (A pointer to a function *component* would be a *closure*, a concept not currently supported in C-like languages.) Pointers to class function members correspond exactly to operations in our model of C++: such pointers *cannot* distinguish between individual methods for the same operation (in the case of pointers to virtual function members).[6]

The operation model is summarized in Figure 9. The illustration assumes that class *D* is derived from class *B*, that *f* is virtual, and that class *X* is unrelated by derivation to either class *B* or class *D*. Spelling maps operation names to oper-

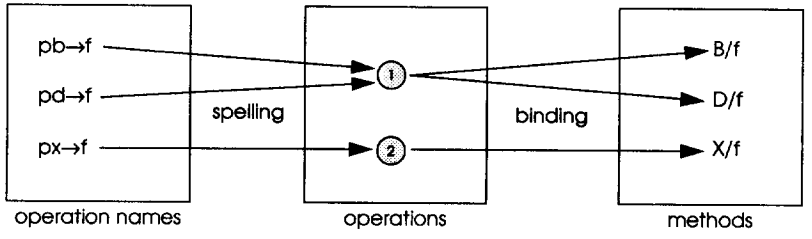

Figure 9. Operations in C++.

6. It is somewhat of an anomaly therefore that pointers to class function members are created using the syntax of explicit qualification, which in invocations is used precisely to make such distinctions!

ations; it involves lexical scoping and overloaded function resolution (both static), as well as evaluation (dynamic) in the case of operation variables (as discussed in the previous paragraph). Binding maps operations to methods, and involves a dynamic lookup based on request parameters (types) in the case of virtual operations (operations corresponding to virtual function members).

Using the access control feature, a C++ class can allow direct client access to a data component of its instances. We can model a client-accessible data component as an operation that returns a pointer to the component. This operation is like a non-virtual member function in having a single method. However, the method is defined by the implementation of C++, not by the class. Thus, use of this feature of C++ makes clients dependent on the implementation of objects.

What are the Values?

The abstract object model specifies that requests include parameters, called *values*. The values denote the information transmitted from the client to the service provider. The question of how C++ function invocations map onto this model of requests deserves some discussion. (Note that we need to be particularly careful in our use of terms in this discussion.)

C++ argument passing is based on these key concepts: A *C++ object* is a region of storage. The contents of a C++ object is an immutable data element called a *C++ value*. Both C++ objects and C++ values are *typed*: the types of a C++ object and its contents are identical. The formal argument of a function is bound upon the invocation of the function to a new local variable, which is a C++ object of the declared type. The initialization of this C++ object is based on information provided by the client in the function invocation. The modeling issue is how to denote this information as a value.

The initialization of a formal argument that is not a class instance is straightforward. (Note that this case includes the case of a formal argument that is a *pointer* to an instance, i.e., an *object name*.) The actual argument expression is evaluated to produce a C++ value of the designated type, which becomes the contents of the formal argument; the evaluation may include an implicit type conversion selected based on the static type of the expression. We model this C++ value as the request parameter; thus, any implicit type conversion is modeled as happening prior to issuing the request.

An alternative model is that the request parameter is the C++ value that is the *input* of the implicit type conversion. This model is viable, but has two disadvantages: (1) The signatures of operations can change over time as new classes with conversion functions are defined. For example, if a new class *C* is defined with a conversion function to *int*, then every operation with a formal argument of type *int* is extended to accept instances of class *C*. (2) Because conversion functions are selected based on static expression types, request parameters would have to encode the static type, which implies use of the multi-object model of derived class instances.

The initialization of a formal argument that is a class instance is more complex. The new class instance is created implicitly, then initialized by a *copy constructor* defined by the class. A constructor is a special kind of invocable entity, similar to a function; an implicit formal argument *this* allows the constructor to initialize the components of a new instance. The copy constructor takes a single argument of type 'pointer to C', where C is the declared class. (A class can define other constructors with different numbers or types of arguments.)

There is a special case, which we refer to below as the *optimized case*. A client can create a temporary object by explicitly invoking a constructor. If the actual argument expression is a constructor invocation, an implementation of C++ has the option of using that temporary object directly as the formal argument object (eliminating the use of the copy constructor), rather than passing a pointer to the object to the copy constructor. In all cases, however, the formal argument object is initialized by a constructor.

Three modeling possibilities come to mind for a formal argument of class type C:

1. The request parameter is a C++ value of type C, which becomes the contents of the formal argument instance. The value can be thought of as being the "output" of the copy constructor (or the explicitly invoked constructor in the optimized case). This choice is appealing because it is analogous to the model for non-instance formal arguments. However, the notion of copying the "output" of the constructor into a *new* instance is inconsistent with the semantics of C++, because the constructor can observe the *identity* of the formal argument instance (using *this*).
2. The request parameter is the pointer argument to the copy constructor, or the pointer to the temporary in the optimized case. This choice is closest to the actual language semantics. It is inconsistent with the model for non-instance formal arguments in that, in the optimized case, the formal argument is modeled as created by the *client* (i.e., prior to issuing the request). There is also a possible confusion between these implicit pointer parameters (for formals of a class type) and explicit pointer parameters (for formals of a pointer type), although this confusion is not important for binding, which examines only the request parameter corresponding to the target object.
3. The request parameter is a structured value that describes the information needed to select and invoke the appropriate constructor(s). The advantage of this choice is that the creation of the new C++ object is local to the service provider in all cases (this model might be the most appropriate for a distributed system based on the C++ object model). The disadvantage is the need to invent a kind of value that has no relation to any C++ type.

Although each choice has disadvantages, we prefer the second alternative (the request parameter is a pointer, i.e., an object name). One effect of this choice is that it conceals the fact that C++ passes instances "by value" when the formal argument is a class type. However, strictly speaking, the only guarantee is that the formal argument is a new instance. The copying of the contents is controlled by the copy constructor, whose behavior can be arbitrary; furthermore, in the optimized case, no copying is performed.

We model default arguments as a notational convenience: the client effectively calls an auxiliary function (associated with the *operation*) to compute the request parameter for an omitted argument. This choice is motivated by the fact that default arguments do not affect the type of a function in C++.

What are the Types?

In the abstract object model, types characterize values that are legitimate request parameters. Based on the previous discussion, a class instance can never be a request parameter, unlike a *pointer* to a class instance. Therefore, we model only C++ non-class types as types in the object model.

The only subtype relation defined over types in our model of C++ is between a type 'pointer to class *D*' and a type 'pointer to class *B*', where *B* is a direct or indirect base class of *D*. This relation is a consequence of the monolithic object model: a value of type 'pointer to class *D*' is legitimate as a parameter in a request that identifies an operation whose signature requires a value of type 'pointer to class *B*'. C++ implicit type conversions (such as the conversion from *char* to *int*) do not define subtype relations, because of our decision to model such conversions as taking place prior to issuing a request.

A C++ class whose members are all pure virtual functions is an *interface*. Such a class defines how certain objects can be used, without constraining how those objects are implemented. A pointer type to such a class is an *interface type*. A C++ function type is an *operation signature*: it can be used to define legitimate request parameter values that identify operations.

A Formal Model of Spelling and Binding

A formal model of spelling and binding in C++ is sketched in Figure 10. This model emphasizes the transformations performed by spelling and binding; it does not attempt to model the semantics of C++. For example, the model does not represent the state of the computational system. Also, the treatment of overloaded function resolution handles only invocations; C++ also resolves overloaded function names in expressions based on context.

The first two cases for spelling handle invocations of explicitly named functions. The second two cases handle invocations using pointers to functions and pointers to class member functions, respectively: we assume that such values are operations (i.e., spelling analogous to the first two cases is performed when pointer values are created). For convenience, we model the type of an argument list as a signature type, for overloaded function resolution. As described above, an expression of a class type evaluates to an object name. The binding model assumes that functions serve as methods, and that nonstatic function members have been transformed to take an explicit *this* argument.

Spelling:

request form	request
f(e1,e2,...)	(op[F[ε,f,Type[ε,(e1,e2,...)]]] E[ε,e1] E[ε,e2] ...)
o.f(e1,e2,...)	(op[MF[Type[ε,o],f,Type[ε,(e1,e2,...)]]] E[ε,o] E[ε,e1] E[ε,e2] ...)
e(e1,e2,...)	(E[ε,e] E[ε,e1] E[ε,e2] ...)
o.e(e1,e2,...)	(E[ε,e] E[ε,o] E[ε,e1] E[ε,e2] ...)

ε ∈ environment
f ∈ identifier
e ∈ expression – identifier
e1, e2 ∈ expression
o ∈ expression (denoting a class instance)

E: environment × expression → value	(evaluation)
F: environment × identifier × signature → function	(lexical scoping and overloaded function resolution)
Type: environment × expression → type	(static type analysis)
MF: class × identifier × signature → function	(class scope lookup for overloaded functions)

op: function → operation
signature ⊂ type
class ⊂ type
function ⊂ value

Binding:

request	method invocation
non-virtual operation:	
(φ v1 v2 ...)	(op^{-1}[φ] v1 v2 ...)
virtual operation:	
(φ n v1 v2 ...)	(VF[φ,n] n v1 v2 ...)

φ ∈ operation
v1, v2 ∈ value
n ∈ object name

op^{-1}: operation → function	
VF: operation × object name → function	(method lookup)
OType: object name → class	(dynamic typing)

VF[φ,n] = the most "specific" function f such that f.class ≥ OType[n] and op[f] = φ

Figure 10. Spelling and binding in C++.

Summary

Our model of C++ objects is summarized by the following points (object model terms are italicized):

- Class instances are *objects*.
- Pointers to class instances are *object names*.
- Pointers reveal *object identity* to clients.
- Base components of derived class instances are not *objects*.
- Public data members correspond to special *operations* that return pointers.
- All function invocations are *request forms*, except for invocations that suppress virtual function lookup, which are *direct method invocations*.
- Except for virtual functions, each function is a distinct *operation*.
- An overriding virtual function is a new *method* for one or more existing *operations*.
- Virtual functions are *generic operations*; overloaded functions are not.
- C++ values of non-class types are *values* (*request parameters*).
- C++ non-class types are *types*.
- *Subtyping* is defined between pointer types based on class derivation.
- A class whose members are pure virtual functions is an *interface*.
- A pointer type to an *interface* class is an *interface type*.
- *Operations* are *values*.
- A C++ function type is an *operation signature*.

Observations

In developing this model of C++ objects, we discovered (as have others) that C++ is a complex language that is difficult to master. Over a period of months, we repeatedly discovered new examples that forced us to reconsider our model. We cannot state with absolute confidence that we have found the last such example! Furthermore, we found several cases whose behavior did not appear to be defined by the existing language definition. We have advised the ANSI C++ committee that a more precise language definition is needed. A specific contribution of the abstract object model is the clear distinction between *operations* (which clients are expected to name) and *methods* (which should be hidden from clients); the existing C++ literature fails to clearly distinguish these concepts, using the term *virtual function* for both (the implementation term *vtable entry* is sometimes used for the *operation* corresponding to a virtual function).

The construction of the C++ object model helped us to identify several problems in the design of C++ itself: the inability of a class definer to prevent client access to overridden methods, the unfortunate inconsistent use of explicit qualification, and the inability to name ambiguous (virtual) operations in a class with multiple base classes.

The C++ object model helps to clarify the aspects of C++ that distinguish it from other object-oriented programming languages:

- Operations in C++ are lexically scoped; a common lexical scope (i.e., a common base class definition) is required for generic operations. (Many object-oriented programming languages, e.g. Smalltalk, define a global name space for operations. Lexically scoped operations are advantageous for programming in the large, as they reduce the probability of accidental name collisions. However, C++ provides inadequate flexibility in naming its lexical scopes to take full advantage of this feature: at the top level, class names share a single global name space. The disadvantage of lexically scoped operations is the need to share a common class definition to permit communication between modules, such as between a client module and an object implementation module; the typical *implementation* of C++ exacerbates this situation by requiring recompilation after most changes to a class definition.)
- An object in C++ defined using multiple inheritance can have inherited parts that are visible to clients as distinct entities. (In most object-oriented programming languages, instances of classes defined using inheritance do not reveal inherited substructure.)
- Using multiple inheritance, a derived class can "link together" operations so that individual methods implement multiple operations.
- Clients in C++ can directly invoke specific methods for an operation. (In many object-oriented programming languages, specific methods cannot even be named. Method combination in such languages is performed using special syntax, such as *super* in Smalltalk. The ability to invoke specific methods is less a concern than the inability to control such access.)
- C++ ordinary function members are operations associated with objects that have exactly one method each.
- In C++, a class instance can have state variables that are directly accessible to clients.
- C++ supports overloaded functions, based on static type analysis. Overloaded functions are a naming convenience.
- C++ request forms are transformed by the insertion of client-specific type conversions, based on static type analysis.

Related Work

Several researchers have developed formal models of objects. Cook [4] developed a model of inheritance for classical object systems using denotational semantics, which has been used to compare inheritance in several object-oriented languages [2]. Reddy independently developed a similar model [7]. In both models, objects are modeled as records indexed by message keys, which are equivalent to our operations. However, neither Cook nor Reddy discuss the mapping from programming language constructs to operations (spelling).

Wand [11] developed a formal model of objects that is closer in breadth to our work. Although there are similarities between the two models, there are significant differences. Wand's model of object includes client-visible state, called attributes. More significantly, Wand's model lacks the concepts of request, operation, and binding. Instead, objects change state in response to other state changes, as specified by constraints called laws. Wand excludes notions like binding and methods as implementation details. Our model intentionally includes these concepts to allow comparison of implementation features (which affect the ability of an object system to support reuse). Wand identifies messages and methods as a source of confusion; our definitions of operation and method resolve this confusion.

Conclusions

We found the process of modeling C++ objects challenging, in part because C++ has many differences from other object-oriented systems. Nevertheless, we conclude that the abstract object model is useful for identifying and explaining the distinctive characteristics of the C++ object model. During the modeling process, several problems in the design of C++ were identified. Although we developed several evaluation criteria during the modeling process, the modeling process remains a subjective one: the ultimate evaluation of a model is its usefulness. This paper has emphasized specific aspects of the abstract object model. Other aspects, such as the model of object implementations, are being developed as we apply the model to additional systems.

References

1. D. G. Bobrow, L. G. DeMichel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices* 23, 9 (1988).
2. G. Bracha and W. Cook. Mixin-based Inheritance. *Proc. OOPSLA/ECOOP-90*, 303-311.
3. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* 17, 4 (Dec. 1985), 471-522.
4. W. Cook. *A Denotational Semantics of Inheritance*. Ph.D. Thesis, Brown University, 1989.
5. M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
6. D. H. Fishman, et al. Iris: An Object-Oriented Data Base System. *ACM Transactions on Office Information Systems* 5, 1 (1987), 48-69.
7. U. S. Reddy. Objects as Closures: Abstract Semantics of Object-Oriented Languages. *Proc. ACM Conference on Lisp and Functional Programming (1988)*, 289-297.
8. A. Snyder. *An Abstract Object Model for Object-Oriented Systems*. Report HPL-90-22, Hewlett-Packard Laboratories, Palo Alto, CA, April 1990.
9. A. Snyder. *The Essence of Objects: Common Concepts and Terminology*. Report HPL-91-50, Hewlett-Packard Laboratories, Palo Alto, CA, May 1991.
10. R. M. Soley, ed. *Object Management Architecture Guide*. Document 90.9.1, Object Management Group, Inc. Framingham, Ma., November 1990.
11. Y. Wand. A Proposal for a Formal Model of Objects. In *Object-Oriented Concepts, Databases, and Applications*. W. Kim, F. H. Lochovsky, eds. ACM Press, 1989, 537-559.