THE O++ DATABASE PROGRAMMING LANGUAGE: IMPLEMENTATION AND EXPERIENCE

Rakesh Agrawal +   *Shaul Dar* * †   *Narain Gehani* *

\+ IBM Almaden Research Center
San Jose, California

† University of Wisconsin
Madison, Wisconsin

* AT&T Bell Laboratories
Murray Hill, New Jersey

ABSTRACT

Ode is a database system and environment based on the object paradigm. The database is defined, queried and manipulated using the database programming language O++, which is based on C++. The O++ compiler translates O++ programs into C++ programs which contain calls to the Ode object manager. The current O++ implementation provides facilities for creating and manipulating persistent objects, and for associatively accessing these objects. We describe the implementation of O++: the Ode object manager, the translation of the database facilities in O++, and our experience.

C++ has emerged as the de facto standard language for software development, and database systems based on C++ have attracted much attention. We provide a detailed description of our implementation with the hope that this paper will serve as a reference for implementors of database systems based on C++.

1. INTRODUCTION

Ode [6, 7, 9, 18] is a database system and environment based on the object paradigm. Ode offers one integrated data model for both database and general purpose manipulation. The database is defined, queried, and manipulated using the database programming language O++, which is an upward-compatible extension of the object-oriented programming language C++ [17]. O++ extends C++ with capabilities suitable for database applications. It provides facilities for creating persistent and versioned objects, organizing persistent objects into clusters, defining and manipulating sets, iterating over sets and clusters, and specifying constraints and triggers.

We have recently completed the implementation of the first release of O++. The current prototype provides facilities for creating and manipulating persistent objects, grouping them into clusters, and associatively accessing the objects by iterating over clusters. The O++ compiler translates an O++ program into a C++ program that contains calls to the Ode object manager library. The library provides facilities for creating and manipulating persistent objects. The translated program is then compiled with the C++ compiler `cfront` and linked with the object manager library to form an executable program (Figure 1).

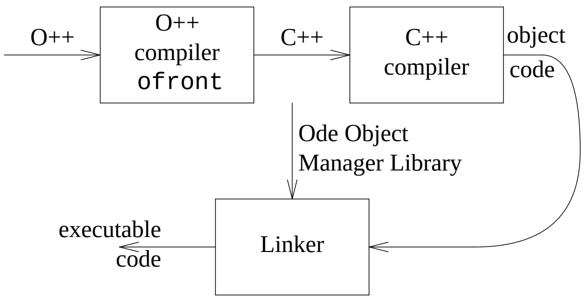


Figure 1. Compilation of An O++ Program

We describe the implementation of O++: the Ode object manager, the translation of the database facilities in O++, and our experience. We discuss the problems we encountered and how we resolved them. C++ has emerged as the *de facto* standard language for software development, and database systems based on C++ have attracted much attention. We provide a detailed description of our implementation with the hope that this paper will serve as a reference for implementors of C++ based database systems. However, our experience should be of interest to the database programming and object-oriented community in general.

Commercial C++ based object-oriented database systems have been implemented amongst others at Object Design, Objectivity, ONTOS, and Versant Object Technology [2-5]. However, only sketchy details of their implementation are available. Closely related to the work presented in this paper is the implementation of the E compiler [28, 29, 31, 33]. Later in the paper, we compare our implementation approach with these and other systems.

The organization of the rest of the paper is as follows. In Section 2, we briefly review O++. The object manager is described in Section 3. In Section 4, we discuss `ofront` and show the translation of O++ programs into C++. In Section 5, we share our implementation experience. We discuss related work in Section 6, and conclude with a summary in Section 7.

2. A BRIEF REVIEW OF O++

We briefly review the features of O++ that have been implemented in the current version of `ofront`. The reader is referred to [6, 7] for complete details of O++. We assume that the reader is familiar with C++ [17].

2.1 Creation and Manipulation of Persistent Objects

The O++ object model is based on the C++ object model as defined by the *class* facility. Classes support data encapsulation and multiple inheritance. O++ visualizes memory as consisting of two parts: volatile and persistent. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary C++ programs. *Persistent* objects are allocated in persistent store and they continue to exist after the program that created them has terminated. Each persistent object is identified by a *unique* identifier, called the object identity. The object identity is referred to as a *pointer to a persistent object*.

Persistent objects are allocated and deallocated in a manner similar to heap objects. Persistent storage operators `pnew` and `pdelete` are used instead of the heap operators `new` and `delete`. Here is an example:

```
persistent Item *pip;
...
pip = pnew Item(initial-values);
```

`pip` is a pointer to a persistent `Item` object; for convenience, we refer to such pointers as persistent pointers (or object ids). `pnew` allocates the `Item` object in persistent store and assigns its id to `pip`. The persistent object pointed to by `pip` can be deleted by

```
pdelete pip;
```

Persistent objects can be copied to volatile objects and vice versa using simple assignments:

```
persistent Item *pip;
Item *ip;
...
*ip = *pip; // copy object pointed to by pip
            // to object pointed to by ip
*pip = *ip; // and vice versa
```

Components of persistent objects are referenced like the components of volatile objects, e.g.,

```
w = pip->weight_kg();
```

2.2 Clusters of Persistent Objects

An Ode database is a collection of persistent objects. Persistent objects of the same type are grouped together into a *cluster*; the name of a cluster is the same as that of the corresponding type, that is, clusters are type extents. Before creating or accessing objects in a cluster, the cluster must first be opened using the macro `copen`, which has the prototype

```
void copen(type-name);
```

2.3 Sets

O++ supports variable size sets. Sets are declared like one-dimensional arrays except that double brackets are used:

```
persistent Item *itemset[[MAX]];
```

2.4 Queries

Values of the elements of a set or a cluster can be accessed using a `for` loop of the form

`for (`*i* `in` *set-or-cluster) suchthat-clause statement*

The loop body, i.e., *statement*, is executed once for each element of the specified set or cluster. In the case of sets, the loop variable *i* is assigned each set element value in turn. In case of a cluster, the loop variable is assigned ids of the objects in the cluster.

The *suchthat-clause* has the form suchthat(e_{st}) and it ensures that every element assigned to *i* satisfies the expression e_{st}.

Here is an example of a `for` loop that prints the names of expensive items:

```
for (pip in Item)
        suchthat (pip->price() > 1000)
    printf("%s\n",(char *)pip->name());
```

3. THE ODE OBJECT MANAGER

The Ode object manager provides low-level facilities for creating and manipulating persistent objects. The object manager library is implemented as a set of C++ classes. Several of these classes are designed as ''template'' (generic) classes (classes parametrized with types) [17]. Due to the unavailability of templates at the time of implementation, the object manager library uses macros to implement generic versions of these classes.

The object manager views the global database as a collection of local databases called ''cluster groups''. A cluster group consists of a set of clusters. Each cluster contains objects of the same type. Objects can point to other objects in the same cluster group.

A cluster group manages two copies of each persistent object: the one on disk and the one in memory (Figure 2). When a program first begins execution, persistent objects (if any) exist only on disk. The process of copying a persistent object to memory is termed *activation*. The cluster group tracks the objects that have been activated and where their memory copies are located. The inverse operation, writing a modified persistent object back to disk is called *passivation* or *synchronization*.

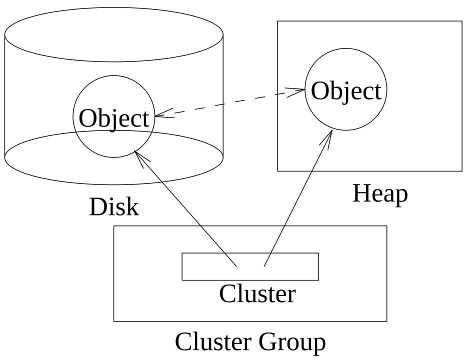
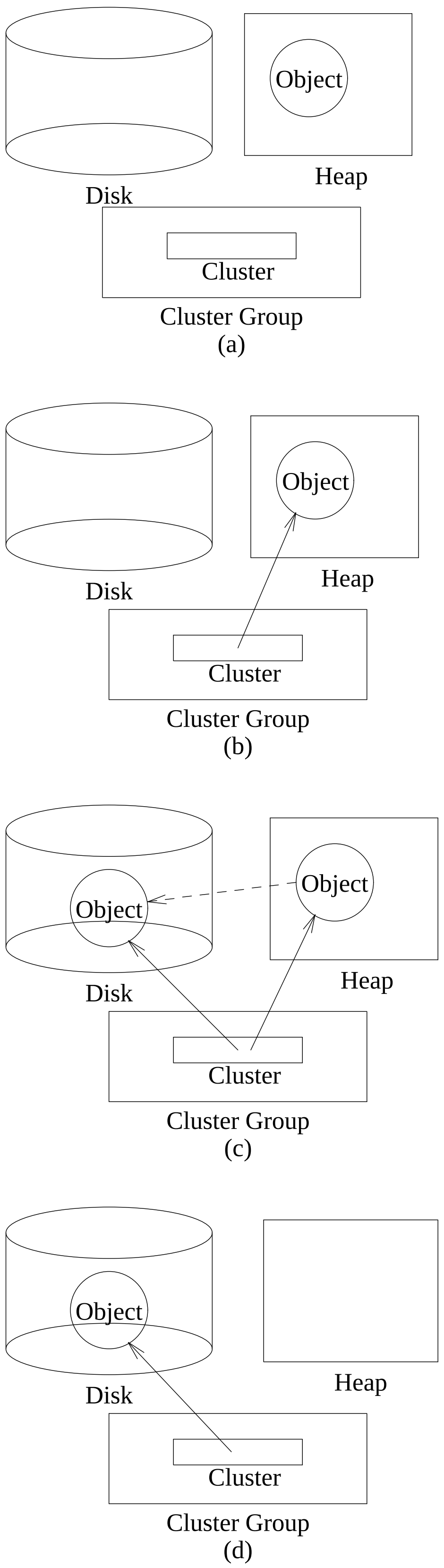


Figure 2. Two Copies of an Object

We now describe the important aspects of the object manager classes. These classes are used by `ofront` in translating O++ constructs. The `ofront` transformations will be discussed in the next section. By necessity, the following description contains low-level implementation details and assumes some familiarity with the basic concepts of C++. It is important to keep in mind that an O++ programmer is not exposed to the interface offered by the object manager. Rather, the declarations and function calls described below are generated by the `ofront`.



3.1 Cluster Groups

Each cluster group constitutes a separate database. An application program can access multiple cluster groups. The object manager implements cluster groups using the class `ClusterGroup`. The user of the object manger library identifies a cluster group by its UNIX® file name. This is the only file name specified explicitly by the user. All other files created by the object manger library are created in the same directory as that containing the cluster group file.

When creating a cluster, the user can specify the cluster group into which the cluster should be placed. There is also a default cluster group with the file name `DCG`. Clusters created without specifying a cluster group are placed in this default group.

Object identifiers, which are used internally to name objects, are allocated on a cluster group basis. The interface to the object manager passes just the cluster group and the object number. Distribution of data over files is strictly up to the object manager.

Class `ClusterGroup` has a member function `sync` that must be called to perform synchronization. Another `ClusterGroup` member function is `deactivate` which causes the cluster group to forget about the memory copies of all activated objects.

3.2 Clusters

A cluster is a persistent collection of objects of a given type. Clusters are implemented using the class template `Cluster`, which is parameterized with the type of the objects that the cluster will contain. `Cluster` is derived from the base class `ClusterBase` that contains all the type independent facilities for implementing clusters.

The object manager creates an object by invoking the `new` operator (Figure 3a). To make the object persistent, it must be inserted into a cluster. The insert operation does not copy the object. Rather, the cluster group saves a pointer to the object (Figure 3b). Eventually, the object is written to disk (Figure 3c) and the memory copy is deleted (Figure 3d).

3.3 Cluster Iterators

Iterator objects are used in object-oriented programming languages to iterate over the elements of a data structure. The Ode object manager library provides a generic iterator class `ClusterIter` that can be used to generate iterators over specific clusters. An iterator class provides functions such as those for getting the next object from a cluster and for resetting

---

* UNIX is a registered trademark of the UNIX System Laboratories.

Figure 3. Creation of a Persistent Object

the iteration. Multiple iterator objects can be associated with the same cluster simultaneously.

A pointer to a *selection* function can be supplied when declaring a cluster iterator. A selection function takes a pointer to an object and returns *true* or *false*. If a selection function is supplied, then the cluster iterator returns only those objects for which the selection function returns *true*.

3.4 Persistent Pointers

A persistent object uses persistent pointers to point to other persistent objects. The object manger library provides several types of persistent pointers that differ in their performance and safety. Class `PersPtr` is the common base class for the different persistent pointer types. O++ persistent pointers (object ids) are implemented as objects of the class `SafePersPtr`. This class is a template class, parameterized with the type of the objects referenced by the persistent pointers. `SafePersPtr` guarantees that the object referenced by a persistent pointer is always activated before it is accessed. It does this by ensuring that the object is activated each time a `SafePersPtr<Type>` object is converted to an ordinary pointer (of type `Type *`) or dereferenced.

Class `PersPtr` and the derived classes implementing the different persistent pointers types provide normal pointer semantics. An ordinary pointer of `Type *` can be assigned to a `PersPtr<Type>` variable, and a `PersPtr<Type>` variable can be used where a `Type *` value is required. Operators `*`, `[]`, `–>`, `++`, `––`, etc., are all defined for the `PersPtr` types.

3.5 Requirements for a Persistent Class

The object manager library requires a class that might have persistent instances to be derived from the special base class `PersBase`.[1] The main reason for this requirement is that when a persistent object is destroyed, the `PersBase` destructor is invoked, and it informs the cluster group that the object no longer exists.

In addition to having `PersBase` as a base class, a persistent class must define three member functions: `diskSize`, `readObj`, and `writeObj`, which compute the disk space required for an object, read an object from disk, and write an object to disk, respectively. These functions are not provided in `PersBase` because they require knowledge of the object type. Unfortunately, C++ does not currently provide object type information at run time.[2] The prototypes of these functions for class `Type` are[3]

```
virtual uint diskSize(void);
static void readObj(ClusterGroup *cgp,
    uint onbr,uint& doff,Type*& objp);
virtual void writeObj(ClusterGroup *cgp,
    uint onbr, uint& doff);
```

where `uint` is a synonym for an `unsigned int`. The `diskSize` function returns the number of bytes of disk space that are needed to write the object to disk (using its `writeObj` function). The disk space required for an object is computed as the sum of the values `sizeof(`*x*`)` for each data member *x* of a pre-defined type plus the sum of the values returned by the `diskSize` member function of each base class and of each data member of a class type.

Function `readObj` is called when activating a persistent object (bringing it from disk to memory) to read its data from disk. The parameters of `readObj` have the following semantics:

- `cgp` — Pointer to the cluster group containing the object. The cluster group provides member functions that the `readObj` function uses to read objects of the predefined C++ types.
- `onbr` — Object number of the object being read.
- `doff` — Disk offset to the start of the object’s image on disk. This value is incremented by the number of bytes read by the cluster group functions that perform the actual I/O before they return.
- `objp` — Pointer to memory location where the object should be assembled.

The `writeObj` function is called to write an object’s persistent data to disk. Its parameters are similar to those of `readObj` and its operation is complementary to `readObj`. Therefore it is not described in detail here.

In order to have a cluster associated with class `Type`, the generic class `Cluster` must be instantiated appropriately. Similarly, in order to have pointers to persistent objects of class `Type`, the generic class `PersPtr` must be instantiated. Macros `ClusterDeclare(Type)` and `PersPtrDeclare(Type)` are used to declare instances of `Cluster` and `PersPtr`. Instantiations of macros `ClusterImplement` and `PersPtrImplement` for class `Type` provide the implementation (body) of the member functions declared in these classes.

4. THE O++ COMPILER

The O++ compiler `ofront` translates each O++ source file into a C++ source file that is in turn compiled into object code by a C++ compiler. The object files are then linked with the object manager library to produce an executable file.

---

1. This requirement implies that the simple types (`int`, `float`, `char`, etc) are not acceptable as arguments to the `Cluster` template. A separate template class `Pers` is used for creating clusters of predefined types.
2. If it did, then these functions could be written once as member functions of the base class `PersBase`. Some object oriented languages (e.g., Smalltalk [19]) do provide this capability. In these languages, a class is itself an object of a (meta) class and an object of such a class makes type information available via its methods (member functions).
3. Functions `diskSize` and `writeObj`, used respectively to compute the disk space required for the object on disk and to write the object to disk, are declared virtual. Therefore, the actual functions invoked will be determined at run time according to the actual object type. Function `readObj` is static and not associated with a particular object, because the object to be read does not yet exist in memory. The last argument for `readObj` serves the role of the implicit `this` pointer in `diskSize` and `writeObj`.

We now describe the transformations that `ofront` applies to an O++ program. The translated code contains calls to the object manager library described in the previous section. The translation is illustrated using the following O++ program:

```
 1 #include <stdio.h>
 2 #include "db.h"  //classes Name & Addr
 3 #include "supplier.h" //class Supplier
 4 class Item {
 5   Name nm;
 6   float wt;
 7   float pr;
 8 public:
 9   persistent Supplier *sup;
10   Item(Name n, float w, float p);
11   Item(Name n, float w, float p,
12               persistent Supplier *s);
13   Name name();
14   float price();
15   double weight_lbs();
16   double weight_kg();
17 };
18 main()
19 {
20   persistent Item *pip;
21   copen(Item);
22   pnew Item("twidleedee", 320, 125);
23   pnew Item("twidleedum", 350, 75);

24   // find the expensive items
25   for (pip of Item)
26       suchthat(pip->price() > 100.00)
27     printf("price of %s is %d\n",
28           (char *)pip->name(),
29           pip->price());
30 }
```

The definitions of classes `Name` and `Supplier` are not given here. We have shown the definition of the class `Item`. This class has three private data members `nm`, `wt`, `pr` and one public data member `sup`. The data member `sup` is pointer to a persistent object, and contains the object id of the supplier of the item. The public interface of `Item` consists of two constructors (lines 10-12) and four member functions (lines 13-16). An appropriate constructor function is automatically called when an object is created, and sets the initial state of the object. We have not shown the bodies of the constructors and the member functions. The member function `name` returns item’s name, the member function `price` returns its price, the member function `weight_lbs` returns its weight in pounds, and the member function `weight_kg` returns its weight in kilos.

The O++ program first opens the `Item` cluster (line 21), it then creates two persistent items using the operator `pnew` (lines 22-23), and finally it iterates over the `Item` cluster to print the names and prices of expensive items costing more than 1000 (lines 25-29).

We now discuss the translation of the various fragments of this O++ program.

4.1 Processing a Class Definition

`ofront` modifies user-defined classes by adding a base class and defining member functions required by the object manager to implement persistence. It also invokes macros to declare instances of the generic classes `cluster` and `PersPtr`, defined in the object manager library, for the user-defined class. Thus, the user-provided definition of the class `Item` is translated by the `ofront` into the following class definition:

```
 1 // declare class PersPtr(Item)
 2 PersPtrDeclare(Item);

 3 class Item : virtual private PersBase {
 4   Name nm;
 5   float wt;
 6   float pr;
 7 public:
 8   SafePersPtr(Supplier) sup;
 9   Item(Name n, float w, float p);
10   Item(Name n, float w, float p,
11         SafePersPtr(Supplier) s);
12   Name name(void);
13   float price(void);
14   double weight_lbs(void);
15   double weight_kg(void);
16   virtual uint diskSize(void);
17   static void readObj(ClusterGroup *cgp,
18            uint onbr, uint& doff,
19            Item*& objp);
20   virtual void writeObj(ClusterGroup *cgp,
21            uint onbr, uint& doff);
22 };

23 ClusterDeclare(Item);
```

In the transformed class definition for `Item`, class `PersBase` has been added to `Item` as a virtual private base class (line 3). Therefore, `Item` now inherits the member functions of `PersBase`. In particular, it inherits the destructor defined with `PersBase`, which is called when a persistent `Item` is destroyed to perform the necessary clean up.

The next change in the transformed definition of `Item` is in the declarations for the pointers to persistent objects. On lines 8 and 11, the type specification `persistent Supplier *` is translated into `SafePersPtr(Supplier)`, which is an instance of the generic class `SafePersPtr` for `supplier`. `SafePersPtr` guarantees that the object referenced by a persistent pointer is always activated before it is accessed.

Finally, three public member functions `diskSize`, `readObj`, and `writeObj` have been generated (lines 16-21). Their bodies are also synthesized and will be described momentarily.

When processing a class definition, `ofront` also invokes macros that declare instances of the generic classes `PersPtr` (line 2) and `Cluster` (line 23) for `Item`, defined in the object library. Creation of these types allows definition of clusters of persistent `Item` objects and pointers to persistent `Item` objects. The declaration of `PersPtr(Item)` prior to the declaration of `Item` serves a role similar to that of a forward declaration of a class: it allows persistent pointers to `Item` to be declared in the definition of `Item`.

The bodies of the `diskSize`, `readObj`, and `writeObj` functions synthesized by `ofront` are as follows:

```
1  uint
2  Item::diskSize(void)
3  {
4    return nm.diskSize()
5           + sizeof(wt)
6           + sizeof(pr)
7           + sup.diskSize()
8           ;
9  }
10 void
11 Item::readObj(ClusterGroup* cgp, uint onbr,
12               uint& doff, Item*& objp)
13 {
14   if (objp == NULL)
15       objp = new Item;
16   Name::readObj(cgp, onbr, doff, &objp->nm);
17   cgp->readObj(onbr, doff, &objp->wt,
18     sizeof(objp->wt));
19   cgp->readObj(onbr, doff, &objp->pr,
20     sizeof(objp->pr));
21   SafePersPtr(Supplier)::
22       readObj(cgp, onbr, doff, &objp->sup);
23 }
24 void
25 Item::writeObj(ClusterGroup* cgp, uint onbr,
26                uint& doff)
27 {
28   nm.writeObj(cgp, onbr, doff) // write nm
29   cgp->writeObj(onbr, doff, &wt, sizeof(wt));
30   cgp->writeObj(onbr, doff, &pr, sizeof(pr));
31   sup.writeObj(cgp, onbr, doff) // write sup
32 }
33 // PersPtr(Item) member functions bodies
34 PersPtrImplement(Item);
35 // Cluster(Item) member functions bodies
36 ClusterImplement(Item);
```

Function `diskSize` computes the disk size of an `Item` by adding the space required for writing its components on disk. For pre-defined component types the `sizeof` operator is used to determine the required space, while for class types the corresponding `diskSize` functions are invoked. Thus, for data items `wt` and `pr`, which are of predefined type `float`, the `sizeof` operator has been used (line 5 and 6), whereas for data items `nm` and `sup` the `disksize` member functions of the corresponding classes have been used (lines 4 and 7). The `disksize` function for `Name` is synthesized when the definition of class `name` is processed by `ofront` and the `disksize` function for `SafePersPtr(Supplier)` is made available by the macro invocation `PersPtrImplement(Supplier)` (see discussion below) which is generated when definition of class `Supplier` is processed.

Functions `readObj` and `writeObj` read an `Item` object from disk and write an `Item` object to disk, by recursively reading and writing components of an `Item` object. For pre-defined types, the class `ClusterGroup`, defined in the object library, provides the `readObj` and `writeObj` functions. These functions have been used on lines 17 thru 20 in `readObj` function and on lines 29 and 30 in `writeObj` function respectively for `Item`. However, for components `nm` (lines 16 and 28) and `sup` (lines 21-22 and 31) `readObj` and `writeObj` functions for their classes are used. As in the case of the `disksize` function, these functions are synthesized when processing their class definitions.

Lines 34 and 36 show the invocations of the macros `PersPtrImplement` and `ClusterImplement` for class `Item`. These calls generate the bodies of the member functions of the instantiations of the template classes `PersPtr(Item)` and `Cluster(Item)`. An example of a synthesized member function of the generated class `PersPtr(Item)` is `SafePersPtr(Item)::disksize()`. An example of a synthsized member function of the generated class `Cluster(Item)` is `Cluster(Item)::insObj`. This function is used to insert a pointer to an `Item` object into an `Item` cluster, as described in the next section.

4.2 Creating Persistent Objects

To create a persistent object, the cluster in which the object will reside must first be opened. In the example O++ program, the `Item` cluster is opened with the O++ statement

```
copen(Item);
```

This statement is translated by `ofront` into the declaration

```
Cluster(Item) _clus_Item("Item");
```

This declaration defines the internally generated variable `_clus_Item` to be of type `Cluster(Item)`. The declaration also causes the constructor for the class `Cluster(Item)` to be called, which in turn opens the `Item` cluster and binds it to the variable `_clus_Item`. Henceforth, operations on `_clus_Item` will apply to the `Item` cluster.

Having opened a cluster, a persistent object can be created and inserted into the cluster using the `pnew` operator:

```
pnew Item("twidleedee", 320, 125, sp)
```

where `sp` is a pointer to a persistent supplier object. This expression is translated by `ofront` into the following expression

```
_clus_Item.insObj(new Item("twidleedee",320,125,sp))
```

in which an object is created (on the heap) by using operator `new` and inserted into the `Item` cluster associated with `_clus_Item` by calling member function `insObj` (see Figures 3a, 3b).

4.3 Queries

O++ queries over objects in a cluster are specified using `for` loops. `ofront` synthesizes a function for the predicate expression specified in the `suchthat` clause. `ofront` also generates an iterator object for iterating over a cluster.

Consider the query:

```
persistent Item *pip;
...
for (pip in Item) suchthat(pip->price() > 1000)
    printf("price of %s is %d\n",
        (char *)pip->name(), pip->price());
```

The declaration for `pip` is translated into

```
SafePersPtr(Item) pip;
```

The `for` loop is translated to the following C++ code:

```
// create iterator object and initialize it
ClusterIter(Item) _iter_Item_pip_1(_clus_Item);
_iter_Item_pip_1.reset(_predicate_1);

while (pip = _iter_Item_pip_1()) {  // iterate
    printf("price of %s is %d\n",
        (char *)pip->name(), pip->price());
}
```

`ClusterIter` is a generic class defined in the object library for iterating over clusters. The definition of class `ClusterIter(Item)` and the bodies of its member functions are generated by the generic macro instantiations `ClusterDeclare` and `ClusterImplement` for class `Item` that are emitted when the definition of class `Item` is processed.

The first statement in the above code creates the iterator object `_iter_Item_pip_1` to iterate over the `Item` cluster (`_clus_Item` is bound to this cluster). The next statement initializes the iteration by calling the member function `reset` defined with the generic class `ClusterIter`. The `reset` function is provided a pointer to an internally generated predicate function `_predicate_1`. This Boolean function evaluates the expression specified in the `suchthat` clause. The next statement is a `while` loop that is executed as long as the iterator `_iter_Item_pip_1` finds an object in the `Item` cluster for which the function `_predicate_1` returns true.

The following is the body of the function `_predicate_1`:

```
uint _predicate_1(Item *pip)
{
    return(pip->price() > 1000);
}
```

Instead of generating a predicate function and passing it as argument to the iterator, it would have been simpler to add the `suchthat` expression as another conjunct in the `while` loop, i.e.,

```
_iter_Item_pip_1.reset();

while (pip = _iter_Item_pip_1() &&
      pip->price() > 1000) {
    printf("price of %s is %d\n",
        (char *)pip->name(), pip->price());
}
```

However, we felt that it is important that the predicate be evaluated ‘‘as close to the data as possible’’. By evaluating the predicate in the object manager rather than in the application program, we can take advantage of the object manager’s knowledge about the accessed data, such as the existence of indices and physical clustering, and make future optimizations easier to implement. This will be especially valuable in the context of a ‘‘client-server’’ architecture.

5. EXPERIENCE

O++ is C++ with a few extensions. We therefore decided to base `ofront` on the C++ compiler `cfront`, because we wanted to avoid duplicating the work done by `cfront`. We considered two alternative strategies for implementing the `ofront`:

1. Extend `cfront` to accept O++ and generate C code. For example, the E compiler [28] implemented at the University of Wisconsin follows this approach.
2. Extend `cfront` to accept O++ and modify it to generate C++ code.

The first alternative would have been simpler to implement — we would only need to make `cfront` recognize the O++ extensions and generate the appropriate code. It would not be necessary to worry (at least not in detail) about what `cfront` (about 40K lines of C++ code) was doing with respect to the C++ part of an O++ program.

However, we chose the second alternative to facilitate interfacing with the Ode object manager library that we decided would be written in C++. There were three main reasons for writing the object manager in C++ instead of in C:

1. C++ would simplify the task of writing the object manager.
2. C++ programmers could use the object manager directly by calling library functions [10].
3. Code generated by `ofront` to use the object manager interface would be much simpler.

5.1 Using C++

Multiple inheritance, as supported by C++, is essential for our implementation. O++ classes are transformed into C++ classes, which amongst other things, have class `PersBase` as a base class. Without multiple inheritance we would not have been able to make `PersBase` a base class of the translated versions of O++ classes that are derived from other classes. C++’s operator overloading turned out to be very useful. By overloading the operators `->`, `*` and `[]` we were able to define class `SafePersPtr` in a way that allowed its objects to be manipulated just like normal pointers. As a result, only the declaration of pointers to persistent objects had to be modified — the code using such pointers did not have to be modified.

5.2 Problems Encountered when Implementing O++ and their Resolution

We now describe the major problems encountered while implementing O++ and our solutions. Although some of these problems may appear to be more programming language issues than strictly database issues, they arise when persistence is added to programming languages. We discuss these problems because the discussion will help those interested in implementing similar C++ based database systems.

5.2.1 Hidden Implementation Pointers: C++ objects, in the translated code produced by most C++ compilers, contain ‘‘hidden’’ pointers that implement inheritance related facilities, such as virtual functions and virtual base classes. In C++, pointers to objects of a type T can also point to objects of types derived from T. Therefore, virtual function invocation must be resolved at run time. The C++ compiler makes this possible by generating for each class a virtual function table (vtable), which stores the addresses of the functions that should be invoked through an instance of that class. When a constructor is called to create a new class instance, it also stores in the new object pointers to the vtables of its class (and its base classes). Invocation of a virtual function is transformed by the compiler to an indirect call that uses the virtual pointers to get the appropriate vtable entry.

Virtual base classes are a facility for specifying a shared base class. Only one copy of a virtual base class subobject appears in derived class object. Instead of each additional copy, a pointer to the base class subobject, called a vbase pointer, is stored.

The vtable and vbase addresses stored in an object are different for each execution of a C++ (and therefore an O++) program. This fact is of no consequence in C++ programs, but in an O++ program, a persistent object can outlive the program that created it. If we save the vtable and vbase pointers when passivating an object on disk and restore them when the object is activated in a different execution of the program (or another program), the pointers will be invalid since they were set in a previous execution of an O++ program. It is the responsibility of the `ofront` to ensure that the vtable and vbase pointers are set correctly. The solution to this problem has two parts;

1. Function `readObj` allocates space in memory for the object being read, by using the `new` operator. The `new` operator invokes the appropriate constructor function to initialize the object. Thus vtable and vbase pointers are initialized properly.
2. Function `readObj` copies an object from disk into memory component by component (and recursively), rather than copying the whole object. The object is also written component-wise. Volatile pointers are neither written nor read. Thus, the vtable and vbase pointers set by the `new` operator are not overwritten.

The hidden pointer problem was also identified in Vbase [1] and E [28]. The approach taken in Vbase was to make the vtables persistent objects. In E, the compiler generates a unique type tag for every ‘‘dbclass’’ having virtual functions, and every instance of such a class contains this tag. The vtables remain volatile objects, but there is a main-memory global hash table for mapping type tags to vtable addresses, which is initialized at program startup. In addition, vbase pointers are replaced in E by an offset, which by definition is a persistent value.

5.2.2 Multiple Definitions of Member Functions Bodies: As described in section 3, the Ode object manager requires that the definition of a class whose objects are to be persistent must include the member functions `diskSize`, `readObj`, and `writeObj`. In addition, macros `ClusterImplement` and `PersPtrImplement` must be invoked with the appropriate value for the `Type` parameter to generate bodies of the member functions for the classes `Cluster(Type)` and `PersPtr(Type)`.

Now, suppose we have two O++ source files `x.c` and `y.c`, both of which manipulate persistent objects of type `Type`. These files are to be compiled separately and linked together to produce an executable program. Unfortunately, `ofront` cannot simply generate the bodies of the functions required for persistence of instances of `Type` in the translated C++ version of each file, i.e., `x..C` and `y..C` because that will result in these functions being multiply defined.

A similar problem arises in the C++ compiler. The C++ compiler automatically generates default constructors for every class. The compiler may have to process the same class definitions multiple times if the same class is included in multiple source files which make up one program. It must, therefore, ensure that the constructor bodies are generated only once, and not every time it sees the class definition. The C++ compiler solves this problem by generating the default function bodies in the source file containing the body of the first explicitly specified non-inline constructor function of the class. Some variations are used to handle the special case when all the explicitly specified constructors are defined inline within the class specification. For example, the AT&T C++ compiler does not generate the default constructors for classes that do not have an explicitly specified constructor.

We could not use the C++ scheme for generating bodies of the member functions because `ofront` must generate the member function bodies discussed above even in the absence of explicitly specified constructors. Instead, we solved this problem by emitting the bodies of these member functions in an ‘‘implementation’’ file that is associated with each O++ source file. For our example, these files are named `x.impl..C` and `y.impl..C`. Within these files, the bodies are protected with shields to avoid multiple definitions:

```
#ifndef _Type
#define _Type
bodies of functions implementing persistence for type Type
...
#endif
```

These implementation files are compiled in the link phase. We concatenate the implementation files associated with the object files that were passed to the link phase. The concatenated file is compiled and then linked with the specified object files. The concatenation and the shields ensure that the functions synthesized by `ofront` for class `Type` will not be multiply defined.

5.2.3 Sets: Many database programming languages provide data structures not provided in the base language. For example, O++, E [28], and ObjectStore’s extended C++ interface [23] provide set type constructors, which are not provided by C++. O++ sets are implemented by translating them into objects of class `Set`:

```
class Set {
    int size; //size of an element
    int max; //no. elem. allocated
    int num; //current no. of elem.
    char *objects; //ptr to elements
public:
    ...
};
```

Set operations are translated into `Set` member and ‘‘friend’’ function calls. For example, the O++ set membership test `e member S`, where `S` is a set and `e` is a possible element of `S`, is translated to the call

```
member(S, e)
```

We did not implement `Set` as a template class. If we had done that, the user of `Set` would have to create an instantiation of the generic `Set` for each element type. This instantiation would include member function definitions customized for the set element type. Unfortunately, emission of these definitions is a problem if the set is local, that is, it is declared inside a function.

C++ does not allow function bodies to be specified within another functions. And these function bodies cannot be emitted outside the nesting function because C++ does not provide facilities for referencing such functions.

Our implementation of class `Set` does not know about the set element types. It treats set elements as sequences of bytes and uses the element size information to manipulate them. Type checks and conversions are performed by `ofront`. Because the element type is not known *a priori*, the `Set` functions that take an element as an argument, namely, the insertion, deletion, and membership test functions, do not declare the element type. Instead, they use the C++ ellipsis notation to accept a variable number of arguments and arguments whose types are not known at compile time. For example, here is the prototype of function `member`:

```
friend int member(Set s, ...);
```

We extract the second argument to `member` using the C++ variable argument mechanism, `stdarg`.

6. RELATED WORK

Several database systems with object-oriented features have been implemented recently, including DASDBS [30], Exodus [13], Gemstone [11], Iris [34], O_2 [16], Orion [22], Postgres [32], and Starburst [25]. New storage managers have been built to meet the needs of object-oriented database systems. Examples of such storage managers include Exodus [12], Mneme [26], and ObServer[20].

An important consideration in implementing object-oriented systems is the mechanism by which persistent objects residing on secondary storage migrate in and out of main memory during program execution. A related issue is the semantics of object ids. An object id can be ‘‘physical’’ or ‘‘logical’’ [21]. Logical object ids offer greater flexibility in reorganizing data on disk, but require an extra indirection step. Gemstone, Orion, and ObServer, for example, use logical object ids, whereas O_2 and Exodus use physical object ids.

The ‘‘object-faulting’’ model for determining when to move objects from disk to main memory was pioneered by PS-Algol [14]. In this model, different formats are used for pointers to persistent and transient objects. Each pointer dereference is checked to determine the pointer format. If the pointer is in ‘‘persistent’’ format, the object is read into memory and the faulting pointer is ‘‘swizzled’’ [27, 35] to point to the memory address of the object. When an object is written back to disk, any swizzled pointers *in* this object are restored to the persistent format.

E initially departed from this object-faulting model and used a ‘‘load-store’’ model [28]. In this model, calls to the storage manager to perform I/O are scheduled by the compiler: before manipulating persistent data, the program must first load it into a buffer, and when the program terminates, it must release the buffer space. If the data has been written, the program must inform the buffer manager that the data is dirty. The current implementation of E [31] has switched to the object-faulting model. The address of a persistent object comprises a physical object id and an offset. The location of an object in main memory is stored in a data structure called ‘‘user descriptor’’. Access to persistent data is done through an interpreter, the E Persistent Virtual Machine (EPVM), which maps a persistent pointer first into a user descriptor, and then to an address in the buffer pool. A limited form of pointer swizzling has been implemented by the EPVM, which avoids the double indirection for local pointer variables. A later design of object faulting and pointer swizzling in E has been presented in [33].

An O++ persistent pointer is a logical object id that contains, amongst other things, a pointer to the cluster group (database) that the object belongs to, and an object number. Associated with the cluster group is a table that holds the disk address of every object in that cluster group. In addition, this object table stores the memory addresses of the objects currently in the buffer pool (a similar object table was used in Mneme [26]). The object number component of an object id serves as an index into the object table. An object is activated by reading the object from the disk and recording its memory address in the object table. An object is passivated by storing the object on disk and invalidating its memory address.

Commercial C++ based object-oriented database systems have been implemented amongst others at Object Design, Objectivity, ONTOS, and Versant Object Technology [2-5, 23]. Objectivity, ONTOS, and Versant provide C++ library interfaces for application development, whereas Object Design’s ObjectStore supports both library interface and a DML preprocessor interface based on `cfront` [4]. The DML preprocessor recognizes C++ extensions and generates C code. ObjectStore supports a virtual memory mapping architecture to uniformly handle transient and persistent data. It also supports associative queries and indexes.

7. SUMMARY

We have presented the details of the implementation of the O++ database programming language, the problems we encountered, and our resolution of those problems. C++ based object-oriented database systems have attracted large attention and several commercial and research products/prototypes are in different stages of implementation. We hope that the implementation details and our experiences provided here would benefit the object-oriented community in general, and those interested in object-oriented systems based on C++ in particular.

We are currently working on a new implementation of O++, using the experience gained from this first implementation. In particular, we realized that the use of generic (template) classes in the generated code leads to significant code expansion and increases program compile time. This is because we have to instantiate the generic classes for each user-defined class. The object manager used in our new implementation avoids this generic expansion, and has no direct knowledge of the type of the objects it stores. This simpler object manager will not be type safe if used directly. But when used to support the O++ compiler `ofront` by storing persistent O++ objects, this is not a concern because `ofront` performs all type checks before generating calls to the object manager interface.

Release 1.1 of O++, implementing the core functionality of the language, can be obtained from N. Gehani. Continuing work on O++ includes the investigation of performance related issues, such as optimization [24], and the incorporation of new facilities into the language. These include support for large objects, versions [9], constraints and triggers [18].

We are also studying the use of a persistent type catalog to store information about types of objects in an Ode Database. The design of such a catalog is an interesting open research problem.

8. ACKNOWLEDGMENTS

We appreciate the helpful comments of J. Richardson, J. Kiernan and A. Biliris. S. Buroff wrote the Ode object manager library. D. Schuh modified `cfront` to generate C++ instead of C; this modified `cfront` was is as the basis for the O++ compiler. Modification of `cfront` to generate C++ was facilitated by D. DeWitt and M. Carey. J. Gava made significant contributions to the implementation of `ofront`.

REFERENCES

[1] ''Vbase Technical Notes'', ONTOS Inc., Burlington, MA, 1987.

[2] ''ONTOS Object Database (Release 2.0) Data Sheet'', ONTOS, Inc., Burlington, MA, Nov. 1989.

[3] ''Product Profile'', Versant Object Technology Co., Menlo Park, CA, 1990.

[4] ''ObjectStore User Guide, Release 1.1.1 for Unix-Based Systems,'', Object Design Inc,, Burlington, MA, Sept 1991.

[5] ''Objectivity/DB (Release 2.0)'', Objectivity, Inc., Menlo Park, CA, Sep. 1992.

[6] R. Agrawal and N. H. Gehani, ''Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++'', *2nd Int'l Workshop on Database Programming Languages*, Portland, OR, June 1989.

[7] R. Agrawal and N. H. Gehani, ''Ode (Object Database and Environment): The Language and the Data Model'', *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 36-45.

[8] R. Agrawal, N. H. Gehani and J. Srinivasan, ''OdeView: The Graphical Interface to Ode'', *Proc. ACM-SIGMOD 1990 Int'l Conf. on Management of Data*, 1990, 34-43.

[9] R. Agrawal, S. J. Buroff, N. H. Gehani and D. Shasha, ''Object Versioning in Ode'', *Proc. IEEE 7th Int'l Conf. Data Engineering*, Tokyo, Japan, Feb. 1991.

[10] S. J. Buroff and D. Shasha, ''A Persistence Library for C++'', AT&T Bell Laboratories, Murray Hill, New Jersey, 1989.

[11] P. Butterworth, A. Otis and J. Stein, ''The GemStone Object Database Management System'', *Comm. ACM 34*, 10 (October 1991), 64-77.

[12] M. J. Carey, D. J. DeWitt, J. E. Richardson and E. J. Shekita, ''Storage Management for Objects in EXODUS'', in *Object-Oriented Concepts and Databases*, W. Kim and F.H. Lochovsky (ed.), Addison-Wesley, 1989.

[13] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. H. Schuh, E. J. Shekita and S. L. Vandenberg, ''The EXODUS Extensible DBMS Project: An Overview'', in *Readings in Object-Oriented Database Systems*, S. Zdonik and D. Maier (ed.), Morgan Kaufmann, 1990.

[14] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey and R. Morrison, ''Persistent Object Management System'', *Software Practice and Experience 14*, 1 (1984), 49-71.

[15] S. Dar, N. H. Gehani and H. V. Jagadish, ''CQL++: An SQL for a C++ Based Object-Oriented DBMS'', *Proc. of Int'l Conf. on Extending Database Technology*, Vienna, Austria, Mar. 1992.

[16] O. Deux et al., ''The O_2 System'', *Comm. ACM 34*, 10 (October 1991), 34-49.

[17] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[18] N. H. Gehani and H. V. Jagadish, ''Ode as an Active Database: Constraints and Triggers'', *Proc. 17th Int'l Conf. Very Large Data Bases*, Barcelona, Spain, 1991, 327-336.

[19] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1981.

[20] M. F. Hornick and S. B. Zdonik, ''A Shared Segmented Memory System for an Object-Oriented Database'', *ACM Trans. Office Information Systems 5*, 1 (Jan. 1987), 70-95.

[21] S. N. Khoshafian and G. P. Copeland, ''Object Identity'', *Proc. OOPSLA '86*, Portland, Oregon, Sept. 1986, 406-416.

[22] W. Kim, J. F. Garza, N. Ballou and D. Woelk, ''Architecture of the Orion Next Generation Database System'', *IEEE Trans. Knowledge and Data Engineering 2*, 1 (Mar. 1990), .

[23] C. Lamb, G. Landis, J. Orenstein and D. Weinreb, ''The ObjectStore Database System'', *Comm. ACM 34*, 10 (October 1991), 50-63.

[24] D. G. Lieuwen and D. J. DeWitt, ''A Transformation-Based Approach to Optimizing Loops in Database Programming Languages'', *Proc. ACM-SIGMOD 1992 Int'l Conf. on Management of Data*, June 1992, 91-100.

[25] G. M. Lohman, B. Lindsay, H. Pirahesh and K. B. Schiefer, ''Extensions to June Starburst: Objects, Types, Functions, and Rules'', *Comm. ACM 34*, 10 (October 1991), 94-109.

[26] J. E. B. Moss, ''Design of the Mneme Persistent Object Store,'', *TOIS 8*, 2 (April 1990), 103-139.

[27] J. E. B. Moss, ''Working with Persistent Objects: To Swizzle or Not to Swizzle'', COINS Technical Report, May 1990.

[28] J. E. Richardson and M. J. Carey, ''Persistence in the E Language: Issues and Implementation'', *Software—Practice & Experience 19*, 12 (Dec. 1989), 1115-1150.

[29] J. R. Richardson, ''Compiled Item Faulting: A New Technique for Managing I/O in a Persistent Language'', *4th Int'l Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept 1990, 3-16..

[30] H. J. Schek, H. B. Paul, M. H. Scholl and G. Weikum, ''The DASDBS Project: Objectives, Experiences, and Future Prospects'', *IEEE Trans. Knowledge and Data Engineering*, March 1990, 25-43.

[31] D. H. Schuh, M. J. Carey and D. J. DeWitt, ''Persistence in E Revisited — Implementation Experiences'', *Proc. 4th Int'l Workshop on Persistent Object Systems 2*, 1 (September 1990), .

[32] M. Stonebraker and G. Kemnitz, ''The POSTGRES Next-Generation Database Management System'', *Comm. ACM 34*, 10 (October 1991), 78-93.

[33] S. J. White and D. J. DeWitt, ''A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies'', *VLDB 92*, Vancouver, BC, Canada, Aug. 1992..

[34] K. Wilkinson, P. Lyngbaek and W. Hasan, ''The Iris Architecture and Implementation'', *IEEE Trans. Knowledge and Data Engineering*, March 1990, 63-75.

[35] P. R. Wilson, ''Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware'', University of Illinois at Chicago Technical Report UIC-EECS-90-6, Dec. 1990.