Making C++ Objects Persistent: The Hidden Pointers

A. Biliris

AT&T Bell Labs
Murray Hill, New Jersey 07974

S. Dar

AT&T Bell Labs
Murray Hill, New Jersey 07974
&
University of Wisconsin
Madison, WI 53706

N. H. Gehani

AT&T Bell Labs
Murray Hill, New Jersey 07974

1. INTRODUCTION

O++ [4, 5, 9] is a database programming language based on C++ [11, 18]. Amongst other things, O++ provides facilities for making C++ objects persistent. Objects of any class can be allocated on the stack, on the heap, or in persistent store. Objects allocated in persistent store are called persistent objects, and pointers to such objects are called persistent pointers. From the O++ user's point of view, the semantics of persistent pointers are identical to those of volatile pointers. In particular, inheritance related mechanisms, including virtual base classes and virtual functions, should behave the same way for both kinds of pointers.

C++ objects of types that have ''virtual'' functions and ''virtual'' base classes contain volatile (''memory'') pointers. We call such pointers ''hidden pointers' because they were not specified by the user. In the case of virtual functions, the hidden pointer points to a virtual function table that is used to determine which function is to be called. In the case of virtual base classes, the hidden pointers are used for sharing base classes.

O++ objects are C++ objects. The O++ compiler `ofront` generates C++ as its output, and relies on the C++ compiler for implementing C++ semantics for objects. In attempting to preserve C++ semantics, we encountered problems because the hidden pointers inside an object are only valid for the duration of the program that created the object. These pointers become invalid across transactions (or program invocations). C++ implementations were not designed to work with persistent objects. As a result, the hidden pointers must be fixed to have correct values so that C++ semantics are maintained for persistent objects.

In this paper, we describe the hidden pointers problem in detail and show how it can be solved. Our solution is novel and elegant in that it *does not* require modifying the C++ compiler or the semantics of C++. In addition to this solution, we outline two alternatives that we considered, and explain why they were not adopted for the O++ compiler.

We also discuss another problem related to making C++ objects persistent. C++ allows base class pointers to point to a derived class objects. Similarly, in O++ a persistent pointer to a base class may actually point to a persistent object of a derived class. When reading the referenced object into memory, the persistent base class pointer must be adjusted to point to the correct offset within the derived class object. Therefore, when an O++ object is written to disk, some type information must be stored inside the object indicating the object type. This information may be used at a later time to read the object properly and to set the base class pointer to an appropriate location in the object. In addition, when the object is read from disk, the hidden pointers must be initialized to their appropriate values.

C++ has emerged as the *de facto* standard language for software development, and database systems based on C++ have attracted much attention [2, 3, 5, 10, 13, 16]. We hope that the details and techniques presented will be useful to database researchers and to implementors of object-oriented database systems based on C++. We expect the reader to be familiar with C++ [18].

The paper is organized as follows. Section 2 illustrates the main memory layout of objects used by C++. A reader familiar with the C++ implementation may skip this section. Section 3 describes the problems associated with making C++ objects persistent, and sections 4 and 5 present solutions to these problems. Section 6 surveys and compares related work, and section 7 presents our conclusions.

2. VIRTUAL FUNCTIONS AND VIRTUAL BASE CLASSES

To illustrate some key points in the representation of C++ objects, we use four classes whose inheritance relationships are shown pictorially in Figure 1. Class `studEmp` is derived from both `student` and `employee` each of which are in turn derived from `person`:

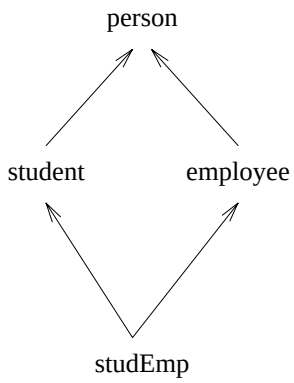

Figure 1. A class hierarchy.

Class `person` is said to be the base class of the classes `student` and `employee`, which are called derived classes. Classes `student` and `employee`, as well as `person`, are the base classes of class `studEmp`.

These types are defined via the C++ type definition facility called the *class*. Class declarations consist of two parts: a specification and a body. The class specification includes the data members (attributes) and member functions (methods) of the class. The class body consists of bodies of functions declared in the class specification but whose bodies were not given there.

2.1 Virtual Functions

Virtual functions are the mechanism used by C++ to support a special kind of polymorphism using ‘‘late’’ binding. C++ allows a base class pointer (or reference) to refer to a derived class object. For example, a `person` pointer (reference) can refer to a `student` object. When a *virtual* function is invoked using this pointer (reference), the specific virtual function that is called depends on the type of the referenced object. It is the task of the C++ compiler to generate code that will invoke the appropriate function.

Assume that classes `person` and `student` are defined as shown below.

```
class person {
public:
    char first[MAX];
    char last[MAX];
    int age;
    virtual void print();
};

class student: virtual public person {
public:
    char university[MAX];
    virtual void print();
};
```

Class `person` defines `print` as a virtual function. Class `student` is derived from (based on) class `person`. The derived class `student` may define its own version of `print`, with a different body (implementation). For example, the body of each `print` function may have been defined as follows.

```
void person::print()
{
    cout << first << " " << last << ", age = " << age << endl;
}
void student::print()
{
    person::print();
    cout << "student at " << university << endl;
}
```

When invoking `print` through a pointer (or reference) to `person`, the actual *virtual* function to be invoked is determined at run time according to the actual type of the referenced object.

As an example, consider the following code:

```
 1  main()
 2  {
 3      person *pp = new person;
 4      student *ps = new student;
 5      …
 6      pp->print();
 7      ps->print();
 8      …
 9      pp = ps;
10      pp->print();
11      …
12  }
```

The first `pp->print` function call (line 6) invokes the function `person::print` because `pp` points to a `person` object. Similarly, the `ps->print` function call (line 7) invokes the function `student::print`. But the second `pp->print` function call (line 10) invokes the function `student::print` even though the type of `pp` is a pointer to `person`. This is because `pp` was assigned a pointer to a `student` object (line 9).

Class `student` declares `person` to be its virtual base class. The keyword *virtual* is used to ensure that only one copy of the virtual base class appears in an instance of the derived class. The virtual base class is shared by all the components of the inheritance hierarchy that specify this class as a virtual base class. Declaring a base class as virtual has no effect with single inheritance but it makes a difference in case of multiple inheritance as we shall see later.

Figures 2 and 3 illustrate the memory representation of objects of type `person` and `student`.

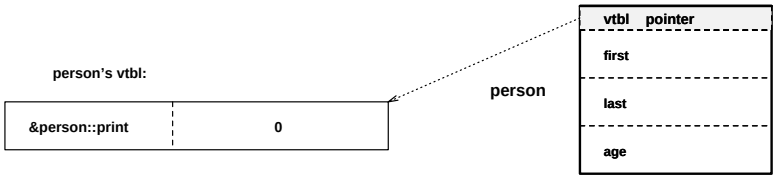

Figure 2. Memory layout of a `person` object

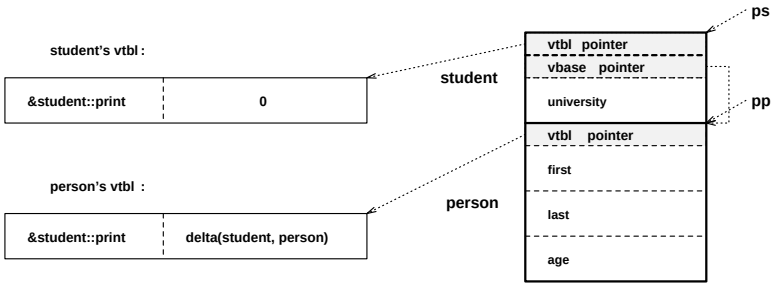

Figure 3. Memory layout of a `student` object

Each object of a class that has virtual functions contains a hidden pointer that points to a virtual function table, called the *vtbl*. The *vtbl* contains addresses of virtual functions. It also contains offsets (deltas), that are used to find the address of a derived class object given the address of a base class sub-object. Returning to our example, after the assignment

```
pp = ps;
```

in line 9, `ps` points to a `student` object, while `pp` points to the `person` sub-object within the `student` object. This is illustrated in Figure 3. Consider the call

```
pp->print();
```

in line 10. The call requires an indirection via the *vtbl* pointer of the `person` sub-object. resulting in the application of the function `student::print`. However, `student::print` expects to get the address of a `student` object as its argument. This address is calculated by subtracting from `pp` the value of `delta(student, person)`, stored in the *vtbl*.

Note that had `print` not been declared as a virtual function in class `person`, then C++ would not have generated the hidden *vtbl* pointer, and calls to the function `print` would not have required any indirection in the translated code.

Because `person` is declared as a virtual base class, references to the `person` component of a `student` object require an indirection through a pointer, called the *vbase* pointer. In this example the indirection may seem unnecessary, but we shall shortly see that it is required to implement sharing of the virtual base class in objects of types specified using multiple inheritance.

2.2 Virtual Base Classes and Multiple Inheritance

When using multiple inheritance, a base class can occur multiple times in the derivation hierarchy. By default, C++ generates multiple copies of such a base class. If only one copy of the base class is to be generated, that is, the base class is to be shared as in other object-oriented languages (e.g., [8, 12, 14]), then the base class must be declared to be a virtual base class.

In the following specification class `employee` is derived from class `person`, and class `studEmp` is derived from both `employee` and `student`.

```
class employee: virtual public person {
public:
    char company[MAX];
    int sal;
    virtual void print();
};

class studEmp: public employee, public student {
public:
    int maxhours;
    virtual void print();
};
```

Because class `person` is a virtual base class of both the `employee` and `student` classes, every `studEmp` object must contain one instance of class `person` instead of two. Both the `employee` and `student` sub-objects share this instance of `person`.

Consider the following code:

```
main()
{
    studEmp *se;
    int a, b;
    …
    se->student::age = a;
    …
    se->employee::age = b;
    …
}
```

Because `se->student` and `se->employee` share the same `person` object, `se->student::age` and `se->employee::age` both refer to the same component, i.e., `se->person::age`.

As before, classes `employee` and `studEmp` may define their own implementation of the `print` function, as shown below.

```
void employee::print()
{
    person::print();
    cout << "employed at " << company << endl;
}
void studEmp::print()
{
    person::print();
    cout << "student at " << university << endl;
    cout << "employed at " << company << endl;
}
```

Figures 4 and 5 illustrate the memory representation of objects of type `employee` and `studEmp`.

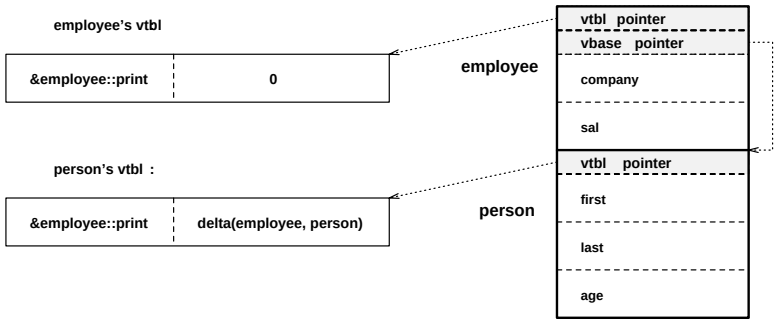

Figure 4. Memory layout of an `employee` object

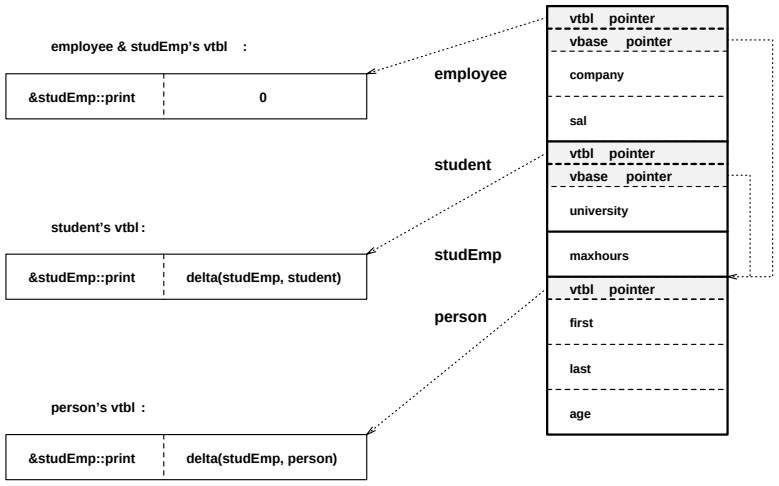

Figure 5. Memory layout of a `studEmp` object

An optimization utilized in most C++ implementations is to share the virtual table of a derived class object with its first non-virtual base class sub-object, since both objects have the same address. This is why in Figure 5, there are 3 virtual tables instead of 4 — `studEmp` and `employee` share the same virtual table.

3. PERSISTENCE AND THE HIDDEN POINTERS PROBLEM

The database programming language O++ [5], which is an upward compatible extension of C++, models its persistent store on the heap. An object allocated on the persistent store becomes persistent. Each persistent object is uniquely identified by its object id (*oid*). A pointer to a persistent object is called a *persistent* pointer, for short. Similarly, a pointer to a volatile object is called a *volatile* pointer and it contains the memory address of the referenced object. Persistent objects are allocated in O++ by using operator `pnew` as opposed to using the operator `new` that allocates volatile objects on the heap. Here is some code showing the allocation of a persistent `employee` object:

```
persistent employee *pe;
...
pe = pnew employee;
```

The type qualifier `persistent` designates pointers to persistent objects. Persistent pointers are used and manipulated much like ordinary pointers.

O++ extends the language constructs provided by C++ so that associative queries over collections of objects can be expressed. For example, here is a code fragment that retrieves high salaried employees (making more than 100K) from the databases and invokes the `print` function on each such employee:

```
for (pe in employee) suchthat(pe->sal > 100000) {
    pe->print();
}
```

3.1 Implementation of Persistence in O++

O++ programs are translated into C++, compiled and linked together with the Ode object manager.

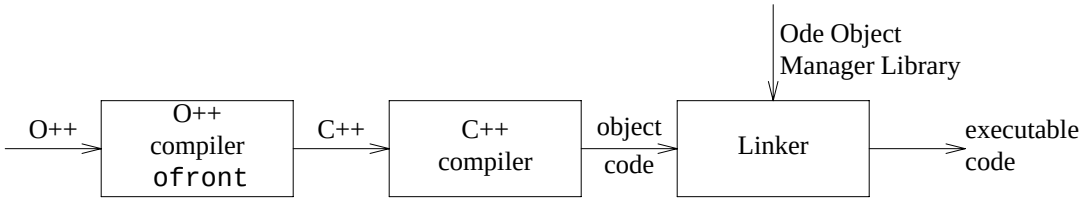

Figure 6. Compilation of An O++ Program

The Ode object manager is a software layer built on top of the EOS storage system [6]. EOS manipulates objects as uninterpreted sequence of bytes stored on disk pages, the unit of I/O. The format of each such object consists of an object header — a tag, followed by the actual length of the object — and then the object itself.[1] The Ode object manager extends the object header to include a pointer to the object's type descriptor; besides that, the on-disk representation of the object is identical to the one used in-memory.

Type descriptors — objects that describe types of objects in the Ode database — are held in the Ode catalog. Catalog information is important to various internal modules of the database system. For example, the query optimizer would access the catalog to check what indexes exist for some collection of objects in order to decide how to execute a selection on it. Here, however, we are only concerned with the components of the catalog that are needed so that an object can correctly be fetched from or placed back on disk. Each entry in the catalog describes a single type. Since every persistent object (including type descriptor objects) has a pointer to its type attached to the object, the Ode object manager can access information about the object's type. In particular, the solutions described in subsequent sections require the object manager to invoke functions specific to each type. These functions are generated by the O++ compiler, which also loads the function addresses into the catalog entry before the main program is executed.

1. If the object is large, i.e., it cannot fit in a single page, then the object is stored in as many pages as necessary to hold the entire object, and a directory to these pages is stored right after the object header [7].

When a persistent pointer is dereferenced, the entire page the referenced object resides on is brought from disk to memory. Once the object is in memory, its starting address is computed and used to reference the object. Thus, the result of the dereference is that we have a C++ object in memory.

3.2 The Hidden Pointers Problem

Virtual functions and virtual base classes have an impact on persistence because of the ''hidden'' *vtbl* and *vbase* pointers (indicated by the shaded areas in the figures shown earlier) generated by C++ compilers to implement these facilities. Virtual function invocations involve an indirection that uses the *vtbl* pointer to access the entries in the virtual function table. And references to the components of virtual base classes must follow the *vbase* pointer. We call the *vtbl* and *vbase* pointers ''hidden'' pointers because they represent implementation related information, and are invisible to the user. Most C++ programmers are not even aware of their existence.

Unfortunately, hidden pointers are volatile pointers, i.e., they are not valid beyond the lifetime of the program that created them. Saving objects containing hidden pointers on disk and then reading these objects back from disk in *another* program means that the hidden pointer values in the objects read from disk are invalid. The same observation holds for the values of data members that are volatile pointers. In the case of pointer members, it is the programmer's responsibility to ensure that the pointers are not used with invalid values. However, in case of hidden pointers it is the responsibility of the system providing persistence, the database programming language O++ in our case, to ensure that the objects read from disk do not contain invalid values prior to their use in the program. Otherwise, a reference to a virtual function or a component of a virtual base class will lead to an illegal memory reference.

4. THE O++ SOLUTION

We now discuss how the O++ implementation handles the hidden pointer problem. We then describe pointer adjustment to allow pointers to persistent objects to also point to persistent objects of derived classes (in accordance with C++ semantics).

As mentioned, the O++ compiler `ofront` generates C++ as its output. The O++ compiler does not have direct access to the hidden pointers. We did not want to modify the C++ compiler to fix the hidden pointers, because this modification would make O++ non-portable. It would require modification of the local C++ compiler, which could affect other C++ programs. We decided to use C++ facilities to place valid values in the hidden pointers.

Our solution is based on the fact that each class constructor, as translated by the C++ compiler, contains code to properly initialize the hidden pointers. This code is executed prior to executing the constructor body, written by the user.

4.1 Using Constructors

The basic scheme is as follows.

1. Read the object from disk. As a result of this request, the page the object resides on is fetched from disk into the buffer pool. Thus, the requested object is now in main memory but it contains bad hidden pointers.
2. Apply a constructor to the object read from disk, to fix the hidden pointers. The constructor must not change the data members of the object.

This solution uses the fact that for every constructor, the C++ compiler adds code to properly initialize the hidden pointers in an object of that type. There are two obstacles to implementing the above scheme. First, C++ does not allow a constructor to be invoked in conjunction with an existing object (as are member functions). However, we can call the constructor indirectly by defining an overloaded version of the global `operator new` function. When an object of class `C` is created by calling `new C`, C++ does two things: (a) calls the function `operator new` to allocate storage for the object.[2] (b) applies an appropriate

constructor (as determined from the arguments to the constructor supplied with the invocation of `new`) to initialize the hidden pointers and components of the object (the latter is as specified by the user).

We do not want to allocate storage for the object. We simply want to make `new` perform the constructor application. Consequently, we overload the `new` operator by defining a new version of `operator new`. We pass to this function the address of the location where we have stored the object read from disk. The function simply returns this address as its result (no storage is allocated).[3] Here is the definition of the overloaded `operator new`:[4]

```
class _ode { };
void* operator new(size_t, _ode *p)
{
    return (void *) p;
}
```

Class `_ode` is a unique type defined to ensure that the overloaded definition of `new` is invoked. Suppose for example that `p` points to an `employee` object that has been read from disk. Then the overloaded definition of `operator new` is invoked as

```
new ((_ode *) p) employee;
```

This invocation of operator `new` invokes the argumentless constructor for class `employee`.

The second obstacle in using this scheme is that we cannot simply invoke a constructor defined by the user to correctly initialize the hidden pointers in the object read from disk because the constructor may modify the values of data members of the object (and even update other objects as well). We need to invoke a constructor that will not modify any data items. That is, it should have a null body.

We first thought of generating for every class a constructor with a null body and a single parameter of type `_ode *`. For example, for class `employee`, this empty constructor would be

```
employee::employee(_ode *) {}
```

This constructor would be invoked if we called operator `new` as illustrated below:

```
new ((_ode *) p) employee((_ode *) 0);
```

Unless otherwise specified, a constructor for a class `D` will invoke the argumentless constructor for each of its base class sub-objects and for every data member of `D` that is a class object. However, the special constructor must

a. invoke the special constructor for each base class sub-object and for every data member of `D` that is a class object;
b. initialize all constant and reference members of `D` in its initializer list.

We abandoned the special constructor solution when we realized that we could not extend it to handle the case when `D` had an array of class objects as a data member. In such a case, C++ *requires* use of the argumentless constructor.

Next we came up with the idea of modifying each user specified constructor so that it would do nothing (execute no statements) when it is called to initialize the hidden pointers. The value of an integer global

---

2. If `C` has an overloaded `C::operator new`, then it is called, otherwise the global `::operator new` is used.
3. The idea of using an overloaded `operator new` to invoke a constructor on an existing object was suggested in a different context in [17].
4. C++ requires the first parameter of an overloaded definition of function `operator new` to be of type `size_t` and that `new` return a value of type `void *`.

variable `_fix_hidden`

```
short _fix_hidden;
```

is used to determine whether or not the constructor was being invoked to fix hidden pointers.

Assume that class `D` defines a constructor of the form

```
D::D(parameter-declarations opt)
{
   ...
}
```

The subscript *opt* indicates an optional item.

This constructor is transformed as follows:

```
D::D(parameter-declarations opt)
{
    if (!_fix_hidden) {
        ...
    }
}
```

This transformation has to be refined to ensure that any initializers present in a constructor definition do not modify any data members. Initializers are given just before the constructor body:

```
D::D(parameter-declarations opt) initializer-list
{
    ...
}
```

Initializers are used to initialize the base class components and the data members of the object. In some cases, initializers are required. For example, if the base class component or a data member can only be initialized by invoking a constructor with arguments, or if the data member is a constant or reference member, then appropriate initializers must be specified.

Initializers that are constructor calls do not have to be modified, because the constructors will have been modified to execute conditionally based on the value of the global variable `_fix_hidden`.

Other initializers, those that specify an initial value for a data member, are modified to change the value of the data member only if the constructor is being called to initialize a newly created object. They have no affect if the constructor is invoked to fix the hidden pointers for an object that has been read from disk. For example, an initializer of the form

```
m(initial-value)
```

where *m* is a data member, is transformed to the initializer

```
m(_fix_hidden ? m : initial-value)
```

When `_fix_hidden` is one, the initializer effectively assigns the member to itself; thus such an initializer does not change the value of the data member.

As an example, a constructor for class `employee` may be defined as follows:

```
employee::employee() : sal(30000)
{
    strcpy(company, "None");
}
```

This constructor is transformed into

```
employee::employee() : sal(_fix_hidden ? sal : 30000)
{
    if (!_fix_hidden) {
        strcpy(company, "None");
    }
}
```

This initialization of hidden pointers is encapsulated in a member function, `reinit`, that is generated for each class. For example, here is the body of the `reinit` function for class `D`:[5]

```
extern short _fix_hidden;
static void D::reinit(void* p)
{
    _fix_hidden = 1;
    new ((_ode *)p) D;
    _fix_hidden = 0;
}
```

Function `reinit` sets the global variable `_fix_hidden` to 1 before invoking the overloaded version of the `new` operator (that does not allocate any storage). Any constructors that are invoked as a result will find `_fix_hidden` to be one, and will not execute any user specified code in the constructor body. The effect of this invocation is simply that the hidden pointers are assigned the right values. Function `reinit` sets the global variable `_fix_hidden` to 0 before returning.

As an example, we give below the code generated by the O++ compiler for class `employee`:

```
class employee: virtual public person {
public:
    char company[MAX];
    int sal;
    virtual void print();
    void reinit(void *);
};

extern short _fix_hidden;

void employee::reinit(void* p)
{
    _fix_hidden = 1;
    new ((_ode *)p) employee;
    _fix_hidden = 0;
}
```

5. `reinit` is declared a `static` member function since it not invoked in association with a particular object.

4.2 Allowing Base Class Pointers to Point to Derived Class Persistent Objects

In C++, a pointer to an object of class `B` can point to an object of a class `D` that is derived from `B`. Similarly, in O++ a pointer to a persistent object of class `B` can point to a persistent object of type `D`. When such a pointer is dereferenced, the object manager brings the object into memory and fixes its hidden pointers, as described above. It then returns a pointer to the `D` object in memory. To conform to C++ semantics, the memory pointer returned must be adjusted properly. In our example, the pointer should be adjusted to point to address of the `B` base class object within the `D` object.

This adjustment is performed as follows. The object manager consults the catalog entry for the object's type. In our example, this type is `D`. The entry contains a list of base classes for this class, and the correct adjustment for each one. These values are filled in by the O++ compiler when it analyzes the definition of this class. The code generated by the O++ compiler for the dereference informs the object manager of the (declared) type of the persistent pointer being dereferenced, in our example `B`. The object manager thus finds the required offset from a `D` object to it's `B` sub-object, *delta*(`D, B`), and adjusts the returned pointer accordingly.

5. ALTERNATIVE TECHNIQUES

We have also considered two other alternative solutions to the problems addressed in the paper. We outline these solutions and explain why we did not adopt them for the O++ compiler `ofront`.

The following definitions are exported by the object manager:

```
typedef unsigned int uint;
class OID {
    …
};
```

5.1 Using Special Member Functions

For each class, the O++ compiler synthesizes two functions, `readObj` and `writeObj`, used to read and write objects of that class. For example, the prototype of these functions for class `employee` is:

```
void employee::readObj(OID& oid, uint& doff);
void employee::writeObj(OID& oid, uint& doff);
```

Function `readObj` is used to read an object from disk. The object might be contained in a larger object. The object id locates the containing object on disk, and `doff` indicates the offset of the object from the beginning of the containing object. Similarly, function `writeObj` is used for writing a value to disk.

In addition, global functions `::readObj` and `::writeObj` are used to read and write values of built-in types, such as integers or character strings:

```
void ::readObj(OID& oid, uint& doff, void *memp, uint cnt);
void ::writeObj(OID& oid, uint& doff, void *memp, uint cnt);
```

`memp` specifies the location in memory where the value is to be stored, and `cnt` specifies the size of the value.

To read an object from disk the following steps are performed:

1. Allocate an object by calling operator `new`. The hidden pointers are thus set correctly.
2. Read the value of the object from disk by invoking the `readObj` function defined for that object's type. This function does not perform a byte copy of the data from disk. Instead, it reads each data member of the object using the `readObj` function defined for the data member's type (and the global `::readObj` function for simple types).

The object manager does not know about object types — it views objects conceptually as uninterpreted bytes. Therefore, it cannot allocate an object by calling operator `new` directly. We allow the object

manager to invoke operator `new` indirectly by encapsulating object allocation in the member function `newObj` which is generated by O++ and whose address is stored in the catalog. The address of `newObj` is loaded for every class in an application program as part of the program initialization process. Given a persistent object, the object manager can find the address of `newObj` for this object's class by following the pointer from the object to the catalog entry describing its type.

We illustrate this mechanism by showing the translated version of class `employee`, which includes the generated functions `readObj`, `writeObj`, and `newObj` as its members:

```
class employee: virtual public person {
public:
    char company[MAX];
    int sal;
    virtual void print();
    void readObj(OID& oid, uint& doff);
    void writeObj(OID& oid, uint& doff);
    static void *newObj();
};
```

The bodies of the `readobj`, `writeobj` and `newObj` functions of class `employee` are as follows:[6]

```
…
void employee::readObj(OID& oid, uint& doff)
{
    // read base classes
    person::readObj(oid, doff);

    // read members
    ::readObj(oid, doff, objp->company, MAX * sizeof(char));
    ::readObj(oid, doff, objp->sal, sizeof(int));
}

void employee::writeObj(OID& oid, register uint& doff)
{

      // write base classes
      person::writeObj(oid, doff);

      // write members
      ::writeObj(oid, doff, company, MAX * sizeof(char));
      ::writeObj(oid, doff, sal, sizeof(int));
}

void *employee::newObj()
{
    return (void *)new employee;
}
```

This solution assumes that `employee` has an argumentless constructor. C++ generates an argumentless constructor automatically if no constructor has been specified for the class. But it does not generate this

6. It might appear that function `newObj` could be declared inline, but then its address could not be taken and stored in the catalog. Similarly, it would not suffice to store the address of a the operator `new` in the catalog, since the application of this operator indirectly through a pointer does not result in the invocation of a constructor.

argumentless constructor if a constructor has been specified for the class. `ofront` therefore generates an argumentless constructor for the class if the user has explicitly specified one or more constructors but has not specified an argumentless constructor.

This solution has the following disadvantages:

1. Three functions must be synthesized for every class.
2. An object is read component-wise from the disk (and written in the same way). Each component requires another function call.

5.2 Using The Assignment Operator

The basic scheme is as follows:

1. Allocate space for the object and read the object from disk. The object contains bad hidden pointers.
2. Allocate another object. This object contains correct hidden pointers.
3. Assign the object read from disk to the new object.

The default assignment performs member-wise assignment of the components of the source object to the destination object. In particular, the hidden pointers are not copied from the source object to the destination object.

We do not discuss this solution in detail. We rejected this solution for the following reasons:

1. Storage has to be allocated twice for every object (optimization can be used to reduce this number).
2. An assignment is required to fix the hidden pointers.
3. This solution assumes that the assignment operator performs member-wise assignment. These are the semantics of the default assignment operator generated by C++. However, users are allowed to define their own version of the assignment operator. This may invalidate our solution, if the explicitly defined assignment operator does not perform member-wise assignment, or has side-effects.

6. RELATED WORK

The hidden pointers problem was also identified in Vbase [1] and E [16]. The approach taken in Vbase was to make the *vtbls* persistent objects. The E compiler *efront* replaces the virtual base class pointer by an offset, an implementation that will probably be used in future C++ compilers as well. It also generates a unique type tag for every class that can have persistent instances (a ‘‘dbclass’’) having virtual functions. Every instance of such a class contains this tag. The E implementation performs run-time virtual function dispatch by hashing on the type tag. In contrast, when an object is brought into memory O++ uses the pointer to the type descriptor of the object to convert invalid C++ hidden pointers to valid ones.

The solutions presented in this paper have the advantage that they do not require modification of the C++ implementation. In addition, virtual function invocation is as fast as in C++ — it requires a single pointer dereference. The Vbase scheme requires a persistent pointer dereference, while the E scheme involves a hashing operation. The disk representation of O++ objects is the same as the memory layout of corresponding C++ objects, except that at the beginning of each persistent O++ object there is a pointer to the object’s type descriptor (another persistent object). The identical on-disk and in-memory object layout allows for code compatibility of O++ and C++.

Systems that implement pointer swizzling [13, 15, 19] encounter problems similar to those that arise from hidden pointers: different formats must be used to refer to objects in memory and on disk. To provide fast access to the memory version of the object, the object id contained in a persistent object is replaced by the memory pointer (the object id is ‘‘swizzled’’). We must similarly replace the hidden pointers by valid values. However, a key difference between pointer swizzling and fixing the hidden pointers is that the O++ compiler does not know the locations of the hidden pointers. It is the C++ compiler that generates the hidden pointers and their locations within an object depends solely upon the code generated by the C++ compiler.

In ObjectStore [13], the object manager knows the location of the hidden pointers in an object of a given type. [7] In addition, the object manager has a table that maps type names into *vtbl* addresses; the table is created during schema generation time. As a special case of the pointer swizzling mechanism, *vtbl* pointers in persistent objects that are brought into memory are assigned the address of the *vtbl* of their respective types, in a recursive fashion. *vbase* pointers are handled by the usual pointer swizzling mechanism, since a *vbase* pointer contained in a persistent object points to another persistent object, the virtual base class sub-object. In comparison, we note that the O++ solution does not require knowledge of the location of the hidden pointer or the *vtbls*, and is thus more portable. In addition, the complexity of recursively fixing the hidden pointers in base class and member sub-objects is simply transferred to the C++ compiler, by calling a constructor.

The problem of adjusting a base class pointer to a derived class object also arose in the implementation of collections in E [16]. Collections are implemented using EXODUS files. A collection whose objects of some type can contain derived type objects. Therefore, an EXODUS iterator over a collection must return an adjusted pointer. This special case was handled by storing the appropriate offset in the object header.

7. CONCLUSION

Hidden pointers are memory pointers that are generated by C++ compilers in objects whose classes contain virtual functions or virtual base classes. These pointer values are not valid across database applications (transactions). Consequently, the hidden pointers in any object retrieved from the database must be ‘‘fixed’’ before the object can be accessed. We have presented solutions to correctly reinitialize the hidden pointers. Our solutions are elegant in that they do not require modifying the C++ compiler or the semantics of C++.

Object-oriented database systems based on C++ are attracting attention because of the emergence of C++ as the language of choice for software development. We hope that the details and techniques presented by will be useful to database researchers and implementors.

8. ACKNOWLEDGMENTS

We appreciate the helpful comments of Ziv Gigus and H. V. Jagadish.

7. Jack Orenstein, personal communication.

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] 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.

[5] 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.

[6] A. Biliris and T. Panagos, ‘‘The Architecture of the EOS Object Store’’, Tech. Rep. 91-014, Computer Science Dept., Boston Univ. , 1991.

[7] A. Biliris, ‘‘An Efficient Database Storage Structure for Large Dynamic Objects’’, *Proc. IEEE 8th Int’l Data Engineering Conference 1992*, Phoenix, Arizona, February 1992, 301-308.

[8] G. Copeland and D. Maier, ‘‘Making Smalltalk a Database System’’, *Proceedings of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Massachusetts, June 1984, 316-325.

[9] S. Dar, R. Agrawal and N. H. Gehani, ‘‘The O++ Database Programming Language: Implementation and Experience’’, *Proc. IEEE 9th Int’l Conf. Data Engineering*, Vienna, Austria, 1993.

[10] O. Deux, ‘‘The O_2 Database Programming Language’’, *Communications of the ACM*, Sep. 1991.

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

[12] S. E. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989.

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

[14] B. Meyer, *Eiffel: The Language Version 3*, Prentice Hall, 1992.

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

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

[17] B. Stroustrup, ‘‘16 Ways to Stack a Cat’’, *Proc. USENIX Winter Conf.*, 1991.

[18] B. Stroustrup, *The C++ Programming Language (2nd Ed.)*, Addison-Wesley, 1991.

[19] 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.

Making C++ Objects Persistent: The Hidden Pointers

A. Biliris

AT&T Bell Labs
Murray Hill, New Jersey 07974

S. Dar

AT&T Bell Labs
Murray Hill, New Jersey 07974
&
University of Wisconsin
Madison, WI 53706

N. H. Gehani

AT&T Bell Labs
Murray Hill, New Jersey 07974

ABSTRACT

C++ objects of types that have virtual functions or virtual base classes contain volatile (‘‘memory’’) pointers. We call such pointers ‘‘hidden pointers’ because they were not specified by the user. If such C++ objects are made persistent, then these pointers become invalid across program invocations. We encountered this problem in our implementation of O++, which is a database language based on C++. O++ extends C++ with the ability to create and access persistent objects.

In this paper, we describe the hidden pointers problem in detail and present several solutions to it. Our solutions are elegant in that they *do not* require modifying the C++ compiler or the semantics of C++. We also discuss another problem that arises because C++ allows base class pointers to point to derived class objects. C++ has emerged as the *de facto* standard language for software development, and database systems based on C++ have attracted much attention. We hope that the details and techniques presented will be useful to database researchers and to implementors of object-oriented database systems based on C++.